USENIX Conference on Domain-Specific Languages *Santa Barbara, California October 1997*

conference

proceedings

The Conference on Domain-Specific Languages Proceedings

Santa Barbara, California
October 15–17, 1997

Sponsored by
The USENIX Association

USENIX®

The Advanced Computing
Systems Association

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $20 for members and $24 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-880446-89-8

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Conference on Domain-Specific Languages

October 15-17, 1997
Santa Barbara, California

Conference Organizers

Table of Contents

Conference on Domain-Specific Languages

October 15-17, 1997
Santa Barbara, California

Letter from the Program Chair

Dear Colleague:

Today's programmers are designing and building systems of vastly greater scale and complexity than ever before—systems with lifetimes in decades, involving millions of lines of code, implemented over distributed systems, in which no single individual has a complete grasp of the code. To create reliable, scalable, maintainable systems, a software engineer must apply a wide variety of tools and techniques. One of these is the use of domain-specific languages.

Domain-specific languages can be a vehicle for formal analysis and optimization methods; they can act as a bridge between visual interfaces and the underlying computation; they can serve as (possibly executable) modeling and prototyping languages; and they can serve as network service interfaces.

Domain-specific languages can act as scaffolding for the software engineering process (as with architectural description languages) or they may be used directly (as with layout languages such as HTML). Domain-specific languages enforce a separation of concerns, insulating the user from unnecessary detail and severing machine dependencies. Domain-specific languages extend software design. The result is a formalism, a concrete artifact that permits representation, optimization, and analysis in ways that low-level programs and libraries do not.

The purpose of this Conference on Domain-Specific Languages is to concentrate on the unique aspects of DSL design, implementation, and application in order to form a body of literature on domain-specific languages, and to refine the DSL technique.

The program committee and a number of outside reviewers (we thank the following: F. Boussinot, Glenn Bruns, E. Coste-Maniere, Ron Cytron, Silvano Dal Zilio, Thierry Despeyroux, Carlos Puchol, Mouly Sagiv, Davide Sangiorgi, Michael Siff, Robert de Simone, and Loren Terveen) read the 55 submissions carefully and deliberated extensively; because of the high quality of all the papers, it was difficult to arrive at a number corresponding to the available presentation slots. Those 23 which appear here include valuable case studies, surveys, insights in design, techniques for definition, tools for implementation, and studies in alternative and complementary approaches. They were chosen for quality, originality, and relevance.

These papers indicate that domain-specific languages constitute a vital topic in programming language research and software practice. It is our hope that those who participate in DSL '97, either by joining in the meeting itself or by referring to these papers in the future, will find the ideas presented here to be of value in framing solutions to existing problems and suggesting profitable avenues for future exploration.

Sincerely,
J. Christopher Ramming

Service Combinators for Web Computing

Luca Cardelli

Digital Equipment Corporation
Systems Research Center
luca@pa.dec.com

Rowan Davies

Carnegie-Mellon University
School of Computer Science
rowan+@cs.cmu.edu

Abstract. The World-Wide Web is rich in content and services, but access to these resources must be obtained mostly through manual browsers. We would like to be able to write programs that reproduce human browsing behavior, including reactions to slow transmission-rates and failures on many simultaneous links. We thus introduce a concurrent model that directly incorporates the notions of failure and rate of communication, and then describe programming constructs based on this model.

1 Introduction

The World-Wide Web [2] is a uniform, highly interconnected collection of computational resources, and as such it can be considered as forming a single global computer. But, what kind of computer is the Web, exactly? And what kind of languages are required for programming such a computer? Before approaching the second question, we must answer the first. In other words, what is the Web's model of computation?

1.1 Some Kind of Computer

We can quickly scan a checklist of possibilities. Is the Web a Von Neumann computer? Of course not: there is no stored program architecture, and no single instruction counter. Is the Web a collection of Von Neumann computers? Down below yes, but each computer is protected against outside access: its Von Neumann characteristics are not exploitable. Is the Web a file system? No, because there is no universally available "write" instruction (for obvious good reasons). Is the Web a distributed database? In many ways yes: it certainly contains a huge amount of information. But, on the one hand the Web lacks all the essential properties of distributed databases, such as precise data schemas, uniform query languages, distributed coherence, consistent replication, crash recovery, etc. On the other hand, the Web is more than a database, because answers to queries can be computed by non-trivial algorithms.

Is the Web a distributed object system? Now we are getting closer. Unfortunately the Web lacks some of the fundamental properties of traditional (in-memory, or local-area) object systems. The first problem that comes to mind is the lack of referential integrity: a pointer (URL[1]) on the Web does not always denote the same value as it did in a previous access. Even when a pointer denotes the same value, it does not always provide the same quality of access as it did in a previous access. Moreover, these pointers are subject to intermittent failures of various duration; while this is unpleasant, these failures are tolerated and do not negate the usefulness of the Web.

Most importantly, though, the Web does not work according to the Remote Procedure Call (RPC) semantics that is at the basis of distributed object systems. For example, if we could somehow replace HTTP[2] requests with RPC requests, we would drastically change the flavor of Web interactions. This is because the Web communication model relies on streaming data. A request results in a stream of data that is displayed interactively, as it is downloaded. It is not that case that a request blocks until it produces a complete result (as in RPC).

At a more abstract level, here are the main peculiarities of a Web computer, with respect to more familiar computational models. Three new classes of phenomena become observable:

- ***Wide-area distribution.*** Communication with distant locations involves a noticeable delay, and behavior may be location-dependent. This is much more dramatic than the distribution observable on a multiprocessor or a local-area network. It is not possible to build abstractions that hide

1. Uniform Resource Locator [7, 8]
2. The HyperText Transfer Protocol is the Web's communication protocol [4, 7, 8].

this underlying reality, if only because the speed of light is a physical limit.

- ***Lack of referential integrity.*** A URL is a kind of network pointer, but it does not always point to the same entity, and occasionally it does not point at all. This is quite different from a pointer in a programming language.
- ***Quality of service.*** A URL is a "pointer with a bandwidth". The bandwidth of connections varies widely with time and route, and may influence algorithmic behavior.

A Web programmer will need to take these new observables into account. This calls for new programming models, and eventually new languages.

Therefore, there are no good names for describing the computational aspects of the Web. We might as well name such a computer a "Berners-Lee computer", after the inventor of HTTP. The model of computation of the Web is implicit in the HTTP protocol and in the Web's hardware and software infrastructure, but the implications of the interaction of the protocol and the infrastructure are not easy to grasp. The protocol is actually quite simple, but the infrastructure is likely to slow down, speed up, crash, stop, hang, and revive unpredictably. When the Web is seen as a computer, e.g., for the purpose of programming it, it is a very unusual computer.

1.2 Some Kind of Algorithms

What kind of activities can one carry out on such a strange computer? Here is an example of a typical behavior a user might exhibit.

Hal carries out a preliminary search for some document, and discovers that the document (say, a big postscript file) is available at four servers: in Japan, Australia, North America, and Europe. Hal does not want to start a parallel four-way download at first: it would be antisocial and, in any case, it might saturate his total incoming bandwidth. Hal first tries the North American server, but the server is overloaded and slow in downloading the data. So, he opens another browser window and contacts the European server. This server is much faster, initially, but suddenly the transfer rate drops to almost zero. Will the North American server catch up with it in the end? While he waits to find out, Hal remembers that it is night in Japan and Australia, so the servers should be unloaded and the intercontinental link should not be too congested. So he starts two more downloads. Japan immediately fails, but Australia starts crawling along. Now Hal notices that the European download has been totally idle for a few minutes so he kills it, and waits to see who wins out between Australia and North America.

What is described above is an instance of an "algorithmic" behavior that is used frequently for retrieving data. The decisions that determine the flow of the algorithm are based on the observable semantic properties of the Web: load, bandwidth, and even local time. The question is: what language could one use to comfortably program such an algorithm? An important criterion is that the language should be computationally complete with respect to the observable properties of the Web:

Every algorithmic behavior should be scriptable.

That is, if a user sitting in front of (say) a browser carries out a set of observations, decisions, and actions that are algorithmically describable, then it should be possible to write a program that emulates the same observation, decisions, and actions.

1.3 Some Kind of Run-Time System

The Web is one vast run-time system that, if we squint a bit, has many of the features of more conventional run-time systems.

There are atomic data structures (images, sounds, video), and compound data structures (HTML[3] documents, forms, tables, multipart data), as described by various Internet standards. There are pointers (URLs) into a universal address space. There are graph structures (MIME[4] multipart/related format) that can be used to transmit complex data. There is a standardized type system for data layout (MIME media types). There are subroutine calls and parameter passing conventions (HTTP and CGI[5]). There are plenty of available processors (Web servers) that can be seen as distributed objects that protect and dispense encapsulated data (e.g., local databases). Finally, there are some nice visual debuggers (Web browsers).

What programming language features could correspond to this run-time system? What could a "Web language" look like? Let's try to imagine it.

A "value" in a Web language would be a pair of a MIME media type and an appropriate content. (For example, the media type may be image/jpeg and the content

3. HyperText Markup Language [3, 7]
4. Multi-purpose Internet Mail Extension [1, 7]
5. Common Gateway Interface [7]

would be a jpeg-encoded image). These values could be stored in variables, passed to procedures, etc. Note that the contents of such values may be in the process of being fetched, so values are naturally concurrently evaluated. Other kinds of values would include gateways and scripts (see below).

The syntax of a programming language usually begins with the description of the "literals": the entities directly denoting a value, e.g., a numeral or a string. A "media literal" would be a pair of a media type and a URL indicating the corresponding content. Such a literal would be evaluated to a value by fetching the URL content (and verifying that it corresponds to the claimed media type).

A "gateway literal" would be a pair of a Gateway Type and a URL indicating a gateway (e.g., a CGI gateway). The gateway type indicates the parameter passing conventions expected by the gateway (e.g., GET, POST, or ISINDEX) and the media types for the requests and replies. A gateway literal evaluates to a gateway value, which just sits there waiting to be activated.

A gateway value can be activated by giving it its required parameters. The syntax for such an activation would look like a normal procedure call: $g(a_1, ..., a_n)$ where g is a literal, variable, or expression that produces a gateway value, and the arguments are (normally) media values. The effect of this call is to package the arguments according to the conventions of the gateway, ship them through the appropriate HTTP connection, get the result, and convert it back to a value. The final value may be rendered according to its type.

We now have primitive data structures (media literals) and primitive control structures (gateway calls). With this much we can already write "scripts". These scripts could be stored on the Web as Internet Media, so that a script can refer to another one through a URL. The syntax for script calls would be the same as above. Scripts would have to be closed (i.e. no free variables, except for URLs), for security and network transparency.

This is arguably a Web language. The scripts are for the Web (not for a particular operating system or file system) and in the Web (not stored in a particular address space or file). Such a language uses the Web as its runtime system.

1.4 Other Issues

Two major issues remain to be addressed.

The first issue is output parsing. Because of the prominence of browsers and browser-ready content on the Web, the result of a query is almost always returned as an entity of type text/html (a page), even when its only purpose is to present, say, a single datum of type image/jpeg. The output has to be parsed to extract the information out of the HTML. Although the structure of HTML pages is relatively well defined, the parsing process is delicate, error-prone, and can be foiled by cosmetic changes in the pages. In order to make Web programming possible on a large scale, one would need some uniform way of describing the protocol of a gateway and, by extension, of a script. This problem is still unsolved and we will not discuss it here further.

The second issue is the design of control structures able to survive the flaky connectivity of the Web. This is the topic of the rest of the paper.

2 Service Algebra

Suppose we want to write a program that accesses and manipulates data on the Web. An obvious starting point is an HTTP library, embedded in some programming language, that gives us the ability to issue HTTP calls. Each HTTP call can fail with fairly high probability; therefore, error-handling code must be written using the error-handling primitives of the language. If we want to write code that reacts concurrently to network conditions and network failures in interesting ways, then the error-handling code ends up dominating the information-processing code. The error-handling and concurrency primitives of common languages are not very convenient when the exceptional code exceeds the normal code.

An alternative is to try to use high-level primitives that incorporate error handling and concurrency, and that are optimized for Web programming. In this section we introduce such primitives. A *service* is an HTTP information provider wrapped in error-detection and handling code. A service combinator is an operator for composing services, both in terms of their information output and of their error output, and possibly involving concurrency. The error recovery policy and concurrency are thus modularly embedded inside each service.

The idea of handling failures with combinators comes, in a sequential context, from LCF tactics [5].

2.1 Services

A Web server is an unreliable provider of data: any request for a service has a relatively high probability of failing or of being unacceptably slow. Different servers, though, may provide the same or similar services. Therefore it should be possible to combine unreliable services to obtain more reliable "virtual services".

A service, when invoked, may never initiate a response. If it initiates a response, it may never complete it. If it completes a response, it may respond "service denied", or produce a real answer in the form of a stream of data.

In the period of time between a request and the end of a response, the main datum of interest is the "transmission rate", counted as bytes per second averaged over an interval. It is interesting to notice that the basic communication protocol of the Internet does not provide direct data about the transmission rate: this must be estimated from the outside.

2.2 Service Combinators

We now describe the syntax and informal semantics of the service combinators in our language. The combinators were chosen to allow common manual Web-browsing techniques to be reproduced with simple programs.

The syntax for our language is given below in BNF-like notation. We use curly brackets { } for grouping, square brackets [ ] for zero or one occurrences, postfix * for zero or more occurrences, postfix + for one or more occurrences, infix| for disjunction, and simple juxtaposition for concatenation. We use '|' to indicate an occurrence of | in the language itself. For lexical items, $[c_1\text{-}c_2]$ indicates a character in the range $c_1\text{-}c_2$.

Services

S ::=
  url(String) | S_1 *?* S_2 | S_1 '|' S_2 | *timeout(Real, S)* |
  limit($Real_1$*,* $Real_2$*, S)* | *repeat(S)* | *stall* | *fail* |
  index($String_1$*,* $String_2$*)* |
  gateway G (String, {Id=String})*

Gateway types

G ::= get | *post*

Lexical items

String ::= " StringChar "*
StringChar ::=
  *any single legal character other than ' " *
  or one of the pairs of characters \' \" \\
*Id ::= {[A-Z] | [a-z] | [0-9]}**
Real ::= [~] Digit+ [. Digit+]
Digit ::= [0-9]

The basic model for the semantics of services is as follows: a service may be invoked at any time, and may be invoked multiple times. An invocation will either succeed and return a result after some time, or fail after some time, or continue forever. At each point in time it has a rate which is a real number indicating how fast it is progressing.

Basic Service

url(String)

The service *url(String)* fetches the resource associated with the URL indicated by the string. The result returned is the content fetched. The service fails if the fetch fails, and the rate of the service while it is running is the rate at which the data for the resource is being received, measured in kilobytes per second.

Gateways

index(String, $String_1$*)*
gateway get (String, Id_1*=*$String_1$ *...* Id_n*=*$String_n$*)*
gateway post (String, Id_1*=*$String_1$ *...* Id_n*=*$String_n$*)*

Each of these services is similar to the service *url(String)*, except that the URL *String* should be associated with a CGI gateway having the corresponding type (*index*, *get* or *post*). The arguments are passed to the gateway according to the protocol for this gateway type.

Sequential Execution

S_1 *?* S_2

The "?" combinator allows a secondary service to be consulted in the case that the primary service fails for some reason. Thus, the service S_1 ? S_2 acts like the service S_1, except that if S_1 fails then it acts like the service S_2.

Concurrent Execution

S_1 | S_2

The "|" combinator allows two services to be executed concurrently. The service S_1 | S_2 starts both services S_1 and S_2 at the same time, and returns the result of whichever succeeds first. If both S_1 and S_2 fail, then the combined service also fails. The rate of the combined service is always the maximum of the rates of S_1 and S_2.

Time Limit

timeout(t, S)

The *timeout* combinator allows a time limit to be placed on a service. The service *timeout(t, S)* acts like *S* except that it fails after *t* seconds if *S* has not completed within that time.

Rate Limit

limit(t, r, S)

This combinator provides a way to force a service to fail if the rate ever drops below a certain limit *r*. A start-up time of *t* seconds is allowed, since generally it takes some time before a service begins receiving any data.

In our original design, this start-up time was applied to the whole service *S*. We later realized that this design leads to an unfortunate interaction with some of the other combinators. This is demonstrated by the example: *limit(t, r, (S_1 ? S_2))*. The problem here is that if S_1 fails after the first *t* seconds, then S_2 is initiated but is not allowed any start-up time, so quite likely the whole service fails.

This motivates the following semantics. The service *limit(t, r, S)* acts like the service *S*, except that each physical connection is considered to have failed if the rate ever drops below *r Kbytes/sec* after the first *t* seconds of the connection. Physical connections are created by invocations of *url*, *index* and *gateway* combinators.

In general, a rate limit can be described as a function *f* from time to rate, and a combinator *limit(f, S)* could be used; the current combinator could then be defined via a step function. The more general combinator is supported by our semantics, but we decided to adopt the current, simpler, definition.

Repetition

repeat(S)

The *repeat* combinator provides a way to repeatedly invoke a service until it succeeds. The service *repeat(S)* acts like *S*, except that if *S* fails, *repeat(S)* starts again.

Unlike many traditional language constructs, the repeat combinator does not include a condition for terminating the loop. Instead, the loop can be terminated in other ways, e.g., *timeout(t, repeat(S))*.

Non-termination

stall

The *stall* combinator never completes or fails and always has a rate of zero. The following examples show how this can be useful.

timeout(10, stall) ? S

This program waits 10 seconds before starting *S*.

repeat(url("http://www.cs.cmu.edu/~rowan")
? timeout(10, stall))

This program repeatedly tries to fetch the URL, but waits 10 seconds between attempts.

Failure

fail

The *fail* combinator fails immediately. It is hard to construct examples in our small language where this is useful, though we include it anyway for completeness, and because we expect it to be useful when the language is extended to include conditionals and other more traditional programming language constructs.

2.3 Examples

We now show some simple examples to illustrate the expressiveness of the service combinators. It is our intention that our service combinators be included as a fragment of a larger language, so for these examples (and in our implementation) we include some extensions. We use "*let*" to make top-level bindings, and we use "*fun*(x) body" and "function(argument)" for function abstraction and application. It is not completely clear how to define the semantics for these extensions in terms of the service model used above. If we are going to very Web-oriented, then perhaps functions should be implemented as gateways, and bound variables should actually refer to dynamically allocated URLs. Regardless, for the simple examples which follow, the meaning should be clear.

Example 1

url("http://www.cs.cmu.edu/")

This program simply attempts to fetch the named URL.

Example 2

gateway get(
"http://www.altavista.digital.com/cgi-bin/query",
pg="q" what="web" q="java")

This program looks up the word "java" on the AltaVista search engine.

Example 3

url("http://www.cs.umd.edu/~pugh/popl97/") |
url("http://www.diku.dk/popl97/")

This program attempts to fetch the POPL'97 conference page from one of two alternate sites. Both sites are attempted concurrently, and the result is that from whichever site successfully completes first.

Example 4

```
repeat(limit(1, 1, url("http://www7.conf.au/"))) |
(timeout(20, stall) ?
    url("http://www.cs.cmu.edu/~rowan/failed.txt")
```

This program attempts to fetch the WWW7 conference page from Australia. If the fetch fails or the rate ever drops below 1 *Kbytes/sec*, then it starts again. If the page is not successfully fetched within 20 seconds, then a site known to be easily reachable is used to retrieve a failure message.

Example 5

```
let av = fun(x)
    gateway get(
        "http://www.altavista.digital.com/cgi-bin/query",
        pg="q" what="web" q=x)
let hb = fun(x)
    gateway get("http://www.HotBot.com/search.html",
        ... MT=x ... )
            (large number of other parameters omitted)
let avhb = fun(x) av(x) | hb(x)
avhb("java")
```

This program defines two functions for looking up search strings on AltaVista and HotBot, and a single function which tries both concurrently, returning whichever succeeds first. It then uses this function to lookup the word "java", to see which engine performs this task the fastest.

Example 6

```
let dbc = fun(ticker)
    gateway post(
        "http://www.dbc.com/cgi-bin/htx.exe/squote",
        source="dbcc" TICKER=ticker
        format="decimals" tables="table")
let grayfire = fun(ticker)
    index(
        "http://www.grayfire.com/cgi-bin/get-price",
        ticker)
let getquote = fun(ticker)
    repeat(grayfire(ticker) ? dbc(ticker))
getquote("DEC")
```

This program defines two functions for looking up stock quotes based on two different gateways. It then defines a very reliable function which makes repeated attempts in the case of failure, alternating between the gateways. It then uses this function to lookup the quote for Digital Equipment Corporation.

3 Formal Semantics

We now give a formal semantics for the service combinators.

3.1 The Meaning Function

The basic idea of the semantics is to define the status of a service at a particular time u, given the starting time t. Possible values for this status are $\langle rate, r\rangle$, $\langle done, c\rangle$, and $\langle fail\rangle$, where r is the rate of a service in progress, and c is the content returned by a successful service.

The *limit* combinator does not immediately fit into this framework. We handle it by introducing an additional parameter in the semantics that is a function that indicates the minimum rate that satisfies all applicable rate limits, in terms of the duration since a connection was started.

Thus our semantics is based on a meaning function $\mathcal{M}$ with four arguments: a service, a start time, a status time, and a rate limit function.

The meaning function implicitly depends on the state of the Web at any time. Instead of building a mathematical model of the whole Web, we assume that a *url* query returns an arbitrary but fixed result that, in reality, depends on the state of the Web at the time of the query.

A complication arises from the fact that Web queries started at the same time with the same parameters may not return the same value. For example, two identical *url* queries could reach a server at different times and fetch different versions of a page; moreover, two identical gateway queries may return pages that contain different hit counters. For simplicity, to make $\mathcal{M}$ deterministic, we assume the existence of an "instantaneous caching proxy" that caches, for an instant, the result of any query initiated at that instant. That is, we assume that *url(String)* | *url(String)* = *url(String)*, while we do not assume that *timeout(t, stall)* ? *url(String)* = *url(String)* for any $t > 0$.

The meaning function is defined compositionally on the first argument as follows:

$$\mathcal{M}(stall, t, u, f) = \langle rate, 0\rangle$$

$$\mathcal{M}(fail, t, u, f) = \langle fail\rangle$$

$$\mathcal{M}(S_1?S_2, t, u, f) =$$
$$\begin{array}{ll} \mathcal{M}(S_2, v_1, u, f) & \text{if } \mathcal{M}(S_1, t, u, f) = \langle fail\rangle \\ \mathcal{M}(S_1, t, u, f) & \text{otherwise} \end{array}$$
$$\text{where } v_1 = \inf\{v \mid \mathcal{M}(S_1, t, v, f) = \langle fail\rangle\} \quad \text{(i.e. the time at which } S_1 \text{ fails)}$$

$$\mathcal{M}(S_1|S_2, t, u, f) =$$
$$\begin{array}{ll} \langle rate, max(r_1, r_2)\rangle & \text{if } s_1 = \langle rate, r_1\rangle \text{ and } s_2 = \langle rate, r_2\rangle \\ \langle rate, r_1\rangle & \text{if } s_1 = \langle rate, r_1\rangle \text{ and } s_2 = \langle fail\rangle \\ \langle rate, r_2\rangle & \text{if } s_2 = \langle rate, r_2\rangle \text{ and } s_1 = \langle fail\rangle \\ \langle done, c_1\rangle & \text{if } s_1 = \langle done, c_1\rangle \text{ and } (s_2 = \langle rate, r_2\rangle \text{ or } s_2 = \langle fail\rangle \text{ or } v_1 \leq v_2) \\ \langle done, c_2\rangle & \text{if } s_2 = \langle done, c_2\rangle \text{ and } (s_1 = \langle rate, r_1\rangle \text{ or } s_1 = \langle fail\rangle \text{ or } v_2 < v_1) \\ \langle fail\rangle & \text{if } s_1 = \langle fail\rangle \text{ and } s_2 = \langle fail\rangle \end{array}$$
$$\begin{array}{l} \text{where } s_1 = \mathcal{M}(S_1, t, u, f) \\ \text{and } s_2 = \mathcal{M}(S_2, t, u, f) \\ \text{and } v_1 = \inf\{v \mid \mathcal{M}(S_1, t, v, f) = \langle done, c_1\rangle\} \\ \text{and } v_2 = \inf\{v \mid \mathcal{M}(S_2, t, v, f) = \langle done, c_2\rangle\} \end{array}$$

$$\mathcal{M}(timeout(v, S), t, u, f) =$$
$$\begin{array}{ll} \mathcal{M}(S, t, u, f) & \text{if } u - t < v \\ \langle fail\rangle & \text{otherwise} \end{array}$$

$$\mathcal{M}(limit(v, r, S), t, u, f) = \mathcal{M}(S, t, u, g)$$
$$\begin{array}{l} \text{where } g(v') = max(f(v'), h(v')) \\ \text{and } h(v') = \\ \quad \begin{array}{ll} 0 & \text{if } v' < v \\ r & \text{if } v' \geq v \end{array} \end{array}$$

$$\mathcal{M}(repeat(S), t, u, f) =$$
$$\begin{array}{ll} \langle rate, 0\rangle & \text{if } u \geq v_n \text{ for all } n \geq 0 \\ \mathcal{M}(S, v_n, u, f) & \text{if } v_n \leq u < v_{n+1} \end{array}$$
$$\begin{array}{l} \text{where (with } \inf\{\} = \text{infinity)} \\ \quad v_0 = t \\ \quad v_{m+1} = \inf\{v \mid v \geq v_m \text{ and } \mathcal{M}(S, v_m, v, f) = \langle fail\rangle\} \end{array}$$

$$\mathcal{M}(url(String), t, u, f) =$$
$$\begin{array}{ll} \langle done, c\rangle & \text{if a connection fetching the URL String at time } t \text{ succeeds} \\ & \text{before time } u \text{ with content } c. \\ \langle fail\rangle & \text{if there exists } u' \text{ s.t. } u' \geq t \text{ and } u' \leq u \text{ and a connection fetching} \\ & \text{the URL String at time } t \text{ fails at time } u' \text{ or has rate } r' \text{ at time } u', \\ & \text{with } r' < f(u'-t) \\ \langle rate, r\rangle & \text{otherwise, if a connection fetching the URL String at time } t \\ & \text{has rate } r \text{ at time } u \end{array}$$

The semantics for *gateway* is essentially the same as for *url*.

A basic property of this semantics, which can be proven by structural induction, is that if $\mathcal{M}(S, t, u, f) = R$, with $R = \langle fail \rangle$ or $R = \langle done, c \rangle$ for some c, then for all $u' \geq u$, $\mathcal{M}(S, t, u', f) = R$.

3.2 Algebraic Properties

Our semantics can be used to prove algebraic properties of the combinators. In turn, these properties could be used to transform and optimize Web queries, although we have not really investigated these possibilities.

We define:

$$S = S' \quad \textit{iff} \quad \forall t, u{\geq}t, f.\ \mathcal{M}(S, t, u, f) = \mathcal{M}(S', t, u, f)$$

Simple properties can be easily derived, for example:

$fail\ ?\ S = S\ ?\ fail = S$
$fail \mid S = S \mid fail = S$
$stall\ ?\ S = stall$
$S\ ?\ stall = S \mid stall$

An interesting observation is that our semantics equates the services *repeat(fail)* and *stall*. However, it is still useful to include *stall* in the language, since the obvious implementation will be inefficient for the service *repeat(fail)*. Conversely, we could consider eliminating *fail* in favor of *timeout*(0, *stall*).

$stall = repeat(fail)$
$fail = timeout(0, S)$

A range of other equations can be derived:

$(S \mid S) = S$
$(S_1 \mid S_2) \mid S_3 = S_1 \mid (S_2 \mid S_3)$
$(S_1\ ?\ S_2)\ ?\ S_3 = S_1\ ?\ (S_2\ ?\ S_3)$
$repeat(S) = S\ ?\ repeat(S) =$
$\quad repeat(S\ ?\ S) = repeat(S\ ?\ \ldots\ ?\ S)$
$repeat(S) = repeat(S)\ ?\ S'$
$timeout(t, limit(u, r, S)) = timeout(t, S) \quad \text{if } t \leq u$
$limit(t, r, S \mid S') = limit(t, r, S) \mid limit(t, r, S')$
$limit(t, r, S\ ?\ S') = limit(t, r, S)\ ?\ limit(t, r, S')$
$limit(t, r, timeout(u, S)) = timeout(u, limit(t, r, S))$
$limit(t, r, repeat(S)) = repeat(limit(t, r, S))$
$limit(t, r, stall) = stall$
$limit(t, r, fail) = fail$

Other "intuitive" properties can be checked against the semantics, and sometimes we may discover they are not satisfied. For example, $S \mid S' \neq S' \mid S$, because the | operator asymmetrically picks one result if two results are obtained at exactly the same time.

4 Implementation

We have implemented an interpreter for the language of service combinators, including top-level definitions and functions as used in the examples. This implementation is written in Java [6].

The implementation also provides an easy interface for programming with service combinators directly from Java. Services are defined by the abstract class Service, declared as follows:

```
public abstract class Service {
    public Content getContent(FuncTime tf);
    public float getRate();
    public void stop();  }
```

To invoke a service, the getContent method is passed a function of time which determines the minimum rate for physical connections, exactly as in the formal semantics. At the top-level, the function ZeroThreshold is normally used, indicating no minimum rate. This method returns the content when the service completes, or returns null when the service fails. During the invocation of the service, the current rate of the service can be found using a concurrent call to the getRate method. Also, the current invocation can be aborted by calling the stop method.

The various service combinators are provided as Java classes, whose constructors take sub-services as arguments. For example, the following Java code corresponds to example 3:

```
new Par(
    new Media("http://www.cs.umd.edu/~pugh/popl97/"),
    new Media ("http://www.diku.dk/popl97/"))
```

This technique could also be used to define interfaces for other domain specific languages within general purpose languages. Essentially the technique is to provide an interface to the abstract syntax representation and the interpreter. If the interpreter is object-oriented, then it will be actually built into the abstract-syntax classes as methods. This is more efficient than providing an interface to the whole interpreter using strings, and it avoids parsing and lexing errors at run-time.

In some sense the implementation is only an approximation to the formal semantics because the semantics ignores interpretation overhead. However, it is quite a close approximation, since interpretation overhead is very small for most programs compared to the time for data to be transmitted.

The rate of a basic service is defined to be the average over the previous two seconds, as calculated by samples done five times a second. This appears to give good results in practice, though admittedly it is somewhat ad hoc.

The implementation uses the Sun Java classes to fetch URLs. A small modification was made to them so that failures are correctly detected, since they normally catch failures and instead return an error page.

5 Conclusions and Future Directions

We have shown that a simple language allows easy expression of common strategies for handling failure and slow communication when fetching content on the Web. We have defined such a language based on service combinators, implemented it, and given a formal semantics for it.

Our intention is that our language will be extended to a more powerful Web-scripting language. Such a language would include some common language features such as functions and conditionals. It should also include additional features for Web-programming beyond our service combinators, for example special constructs for manipulating HTML content, possibly linked to a browser. Another direction is to allow scripts themselves to be stored on the Web, and in our implementation we have experimented with this. It should also be possible to write scripts which provide content on the Web, and perhaps even export a function as a CGI gateway. A full Web-scripting language might even allow a thread of execution to migrate via the Web.

A language with all these features would certainly be very powerful and useful. In this paper we have concentrated only on one particular aspect which is unique to the Web, namely its unreliable nature. By first considering the fundamental properties of the Web we have built a small language whose computation model is tailored for Web programming. We hope that this language and model will serve as a firm foundation for larger Web scripting languages. Elements of our language design and formal semantics should also be useful to designers of other domain specific languages in domains which include real-time concerns or where failures are common.

Acknowledgments

Luca Cardelli would like to thank David Jefferson and a group of people at Digital Palo Alto who met to discuss Web-related issues. Those discussions provided the initial stimulus for this work.

Rowan Davies would like to thank Digital SRC, including all its staff, for providing the opportunity to work in a exceptionally stimulating and pleasant environment during the summer of 1996.

Martín Abadi and Paul McJones reviewed early drafts.

References

[1] Borenstein, N., and N. Freed, **MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies**, RFC 1521, Bellcore, Innosoft, September 1993.

[2] Berners-Lee, T., R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret: **The World-Wide Web**. CACM 37(8): 76-82, 1994.

[3] Berners-Lee, T., and D. Connolly, **Hypertext Markup Language - 2.0**, RFC 1866, MIT/W3C, November 1995.

[4] Berners-Lee, T., R. Fielding, and H. Frystyk, **Hypertext Transfer Protocol - HTTP/1.0**, RFC 1945, MIT/UC Irvine, May 1996.

[5] Gordon, M., R. Milner, and C. Wadsworth, **Edinburgh LCF**. Lecture Notes in Computer Science 78. Springer-Verlag. 1979.

[6] Gosling, J., B. Joy, and G. Steele, **The Java Language Specification**. Addison-Wesley. 1996.

[7] Internet Engineering Task Force, **Internet Standards**. The Internet Society, <http: //www.isoc.org>. 1997.

[8] World Wide Web Consortium, **HTTP - Hypertext Transfer Protocol**, <http://www.w3.org/pub/WWW/Protocols/>. 1997.

A Domain Specific Language for Video Device Drivers: from Design to Implementation

Scott Thibault Renaud Marlet Charles Consel

IRISA / INRIA - Université de Rennes 1
Campus Universitaire de Beaulieu
35042 Rennes cedex, France
{sthibaul,marlet,consel}@irisa.fr
http://www.irisa.fr/compose [*]

Abstract

Domain-specific languages (DSL) have many potential advantages in terms of software engineering ranging from increased productivity to the application of formal methods. Although they have been used in practice for decades, there has been little study of methodology or implementation tools for the DSL approach. In this paper we present our DSL approach and its application to a realistic application: video display device drivers.

The presentation focuses on the validation of our proposed framework for domain-specific languages, which provides automatic generation of efficient implementations of DSL programs. Additionally, we describe an example of a complete DSL for video display adaptors and the benefits of the DSL approach in this application. This demonstrates some of the generally claimed benefits of using DSLs: increased productivity, higher-level abstraction, and easier verification. The DSL has been fully implemented with our approach and is available [1].

1 Introduction

In contrast to a general purpose language (GPL), a *domain-specific language* (DSL) is a language that is expressive uniquely over the specific features of programs in a given problem domain. It is often small and declarative; it may be textual or graphic. DSLs have also been called *application domain languages* [6], *little* or *micro-languages* [2], and are related to scripting languages [18]. DSLs have been used in various domains such as financial products [1], telephone switching systems [11, 16], operating systems [19], and robot languages [5]. Languages such as SQL, TeX and shells may also be considered DSLs.

Software architectures based on DSLs primarily aim at achieving faster development of safer applications. Because constructs in a DSL abstract key concepts of the domain, the developer (that does not have to be a skilled programmer) can write more concise and higher level programs in less time. Programming with a DSL also contributes to safety because it is less error-prone than with a GPL. Additionally, high-level constructs translate, in practice, into the reuse of validated components. Moreover, when the language is small and specific, it is possible and easier to build automatic validation and test generation tools. For example, termination properties may be considered if the language is not Turing-complete.

A DSL may also be seen as a way to parameterize a generic application or to designate a member of a program family. A program family is a set of programs that share enough characteristics that it is worthwhile to study them as a whole. In fact, designing a DSL actually involves the same *commonality analysis* [11] that is used in the study of a program family: assumptions that are true for all members of the family and variations among members. This process should be performed both by domain experts and software engineers.

Though actual uses of DSLs record benefits such as productivity, reliability and flexibility [15], implementing DSLs is often difficult and costly [7]. There are two main approaches to language implementation, each with significant disadvantages: those that are based on compilers (translation from

*This work has been partly supported by FRANCE TELECOM contract CNET 96-1B-027 and DARPA contract F19628-95-C-0193.

[1]http://www.irisa.fr/compose/dsl/genprog.html

the DSL to a target machine or GPL) are not easy to write or to extend, and extensions require skills in compiler technology that cannot be expected from "domain developers"; those that are based on interpreters are easier to write or to extend but are less efficient [4]. This issue also impacts maintainability [21] because complexity in the compiler defeats the software engineering goals of using a DSL. Depending on objectives, either one or the other style of implementation is thus chosen: application generator or interpreter.

We have proposed a framework for the development of application generators that reconciles both alternatives [20]. It relies on *partial evaluation* [12, 14], a program transformation technique that is well suited to automatically transform interpreters into compilers [13]. Partial evaluation exploits known information about a program's input to be able to evaluate parts of a program in advance. Given a program and the known portion of its input, a partial evaluator produces a specialized program. In this new semantically equivalent program, computations depending on known values have already been performed.

Our framework is structured into two levels. The first level consists of the definition of an abstract machine, whose operations can be viewed as generic components that capture important operations of the domain. The second level is the definition of a micro-language in terms of the abstract machine operations, thus providing a high level interface to the abstract machine. The use of partial evaluation in our framework is twofold, corresponding to each level: it maps an abstract machine into an efficient implementation, and a micro-program into an abstract machine program. The development of this framework is supported by industry partners for realistic applications.

This paper describes a realistic application of our framework for the automatic generation of video card drivers. This domain naturally forms a program family, for which DSLs are well suited. We present the design and definition of a complete DSL for video display adaptors. Concerning performance, we show how partial evaluation can yield efficient drivers. Concerning safety, we insure that all generated drivers can be proven to terminate and define some analyses that can greatly improve their reliability.

Our contributions can be summarized as follows:

- We validate our framework of application generator design on a realistic example: video card device drivers.

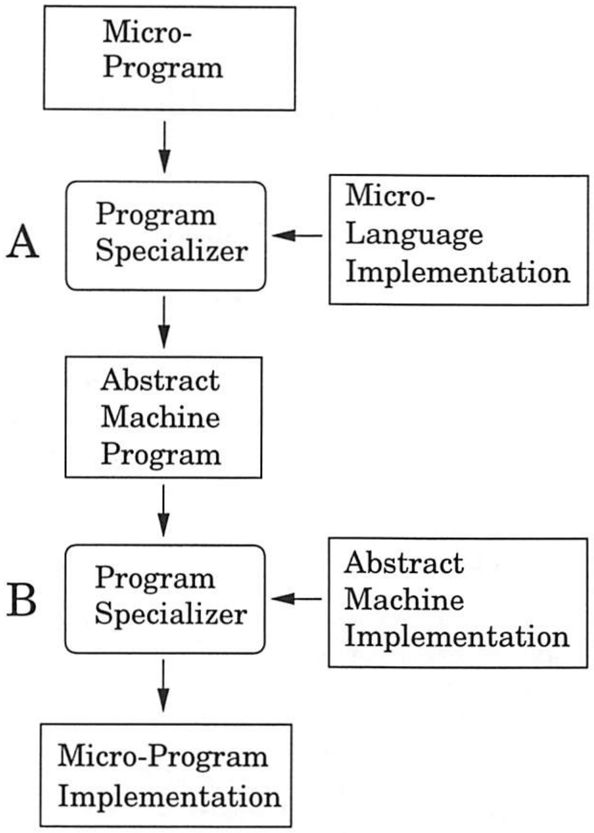

Micro-Program
A
Program Specializer
Micro-Language Implementation
Abstract Machine Program
B
Program Specializer
Abstract Machine Implementation
Micro-Program Implementation

Figure 1: Generic program instantiation.

- We define a DSL for generating such drivers. This restricted language allows program verifications.

- We provide a flexible implementation of this language that generates efficient video drivers.

- We illustrate the benefits of DSLs as a software architecture.

The rest of the paper is organized as follows. Section 2 describes our framework for application generator design in further detail. Section 3 presents the domain of video card drivers. Section 4 describes the two-level design: abstract machine and graphics adaptor language. Section 5 discusses the results of applying this approach to the domain of video drivers. Section 6 summarizes the results of this experiment and identifies future work, both for the language and the framework.

2 The DSL Framework

In a previous paper [20] we presented an approach to generic software design. In this approach, we consider the implementation of a program family as a generic program. The parameterization of this generic program corresponds to variations within the program family and can be represented using a micro-language. A *micro-language* is a small restricted DSL which formally defines the program family (an instance of the program family is specified by a micro-program) and is restricted to allow analysis. The generic program implementing a program family is constructed in two layers, and automatic instantiation is performed by program specialization (i.e., partial evaluation), as illustrated in Figure 1. Together, the two applications of program specialization provide a complete path from a micro-program to an efficient implementation.

The abstract machine is the definition of the fundamental *operations* of the domain that are used to implement members of the program family. The abstract machine is implemented as a highly parameterized library, whose parameters represent *operational* variations within the domain. Any given abstract machine program provides values for operation parameters that indicate the desired functionality. A partial evaluator can eliminate the genericity from the library functions using these known values to produce an efficient implementation, as shown in the bottom section of Figure 1.

The micro-language captures the variations within the program family in terms of *what* family members do, as opposed to *how* family members operate, which is captured by the abstract machine. The micro-language is implemented by an interpreter, which invokes abstract machine operations by calling the corresponding library subprograms. The micro-language provides an interface to the abstract machine, and the interpreter implements a mapping from a micro-program to the abstract machine. This mapping depends only on the micro-program such that, given a micro-program, a partial evaluator, with the micro-program as known input, eliminates all operations in the interpreter leaving only the remaining calls to the abstract machine. Thus, a partial evaluator can produce an abstract machine program from a given micro-program, as shown in the top section of Figure 1.

For the application of this approach we use a partial evaluation system named Tempo [8]. Tempo is a fully automatic partial evaluator for C. Users of Tempo specify inputs to the entry point and global variables as either known or unknown. In our approach, we insure the successful application of partial evaluation via the separation of the abstract machine and the interpreter, each having its on state represented in C by global variables. The interpreter state is specified as known and the abstract machine state is specified as unknown. The following simple rules guarantee correct separation and thus successful automatic partial evaluation.

1. Interpreter subprograms may only use variables of the abstract machine state as actual parameters of a subprogram call.

2. Abstract machine subprograms may not contain any references to the interpreter state.

3 Video Driver Domain

This section introduces the domain of the experiment: video adaptor device drivers. A *video adaptor* (or *video card*) is a hardware component of a computer system which stores and produces images on the display. Video cards consist of a frame buffer, and a graphics controller. The *frame buffer* is a bank of high speed memory used to store the display data, including the currently visible image. The *graphics controller* consists of two main functionalities: producing the video signal for the display, and providing access to the frame buffer to create the display image. Graphics controllers all provide similar sets of functionalities (e.g., changing the display resolution).

Although all adaptors provide similar functionalities, their programming interface is different from vendor to vendor, and often between successive models of the same adaptor. This is true of most devices, and is resolved by the use of device drivers. Device drivers generally consist of a library of functions that implement a standard API that is fixed for all devices. Thus the driver's purpose is to translate the standard API operations into the operations required by a specific device, providing a uniform interface to the operating system and applications.

Video device drivers provide two main services to the operating system and applications. The first is to put the graphics display into different video modes. A *video mode* (or *graphics mode*) is defined by the horizontal and vertical resolution, the number of colors per pixel and screen refresh rates. The second service provided by the driver is to provide access to hardware drawing operations. For example, most video cards provide line drawing hardware, which draws lines on the display at a much faster rate than would be possible in software.

4 Application of the Approach

We have applied the approach described in section 2 to a family of device drivers for video adaptors. We considered an already existing set of device drivers from the XFree86 X Window server created by The XFree86 Project, Inc. [23]. The XFree86 SVGA server is a generic X Window server, written in C, supporting several different cards using a device driver architecture. This server contains drivers for cards from about 25 different vendors. Additionally, each driver supports as many as 24 different models from the same company. This structure alone indicates that there is enough similarities between models of the same vendor to implement them as a generic program, but that it is not reasonable to do so for multiple vendors. This may be due to efficiency, but more likely is due to the lack of a methodology to handle larger scales of variation.

The remainder of this section details the application of our approach to the construction of an application generator of video drivers (for different vendors) for the X Window server. We first discuss the definition of an abstract machine for the domain, identified by studying the existing device drivers. Then we describe a DSL for generating video drivers and related design issues.

4.1 The Abstract Machine

The abstract machine for the video driver domain was designed primarily by studying the implementation of existing drivers. The abstract machine was also iteratively refined during the development of a DSL. We identified three patterns which appeared in the existing drivers that could be used to guide the definition of abstract machine operations.

Operation pattern. The first of these patterns corresponds to simple atomic operations in the abstract machine. There are two forms in which this pattern appears: as repeated fragments of code that differ only by data, and as fragments which perform the same treatment but have a small number of variations on how it is performed. In the first case, the fragments are often already identified and placed in a library or defined as a macro. These fragments directly correspond to abstract machine operations.

As an example of the second case, the device drivers are dominated by occurrences of code fragments which read or write data from or to the video card. Communication with hardware devices can be handled in a small number of different ways, and the scheme chosen varies from vendor to vendor. There were several occurrences of three of these different schemes of I/O, differing only in certain data (e.g., the I/O address). These fragments were captured in a single abstract machine operation defined as follows:

```
write_port(port_number: integer,
           index: integer,
           indexed: boolean,
           pair: boolean,
           pci: boolean)
```

This instruction is parameterized by flags to specify which scheme to use (indexed, paired, or PCI), and the data used by the scheme to perform the I/O (port_number, index).

Combination of operations pattern. The second type of pattern recognized can be identified as expressions or combinations of operations. This pattern is characterized by expressions or combinations of operations that have no commonalities between members of the family. For example, in the device drivers there are sequences of shifts and logical expressions which are different for every driver. Although there are no commonalities in those expressions from one driver to the next, we can identify a sufficient set of operations to construct any instance. The selection of these operations depends not only on the existing samples, but on an understanding of the domain, and speculation on the future of the domain.

The following code fragment shows an example of this pattern from one of the existing drivers.

```
outb(0x3C2, ( inb(0x3CC) & 0xF3) |
            ((no << 2) & 0x0C));
outb(OTI_INDEX, OTI_MISC);
outw(OTI_INDEX, OTI_MISC |
      ((( inb(OTI_R_W) & 0xDF ) |
       (( no & 4) << 3)) << 8));
```

This portion of the driver maps the value of no onto the appropriate registers in order to select the clock. For a given driver, there may be any number of reads, writes, shifts and logic operations, but no other operations. Thus, we can implement any given driver with a sequential composition of a small number of abstract machine operations.

Control pattern. The last pattern consists of code fragments that share a common control structure, but contain code fragments matching the combination of operations pattern previously discussed. For example, consider a function of the device driver which is used to save, restore, and set the clock value on the video card.[2] This function has the following form:

```
Bool ClockSelect(int no)
{
  switch (no) {
  /* Save the clock value. */
  case CLK_REG_SAVE:
    Series of I/Os and logic operations.
    break;
  /* Restore the saved clock value. */
  case CLK_REG_RESTORE:
    A second series of I/Os and logic operations.
    break;
  /* Set the clock value to no. */
  default:
    A third series of I/Os and logic operations.
  }
}
```

The series of I/Os and logic operations in this example follow the combination of operations pattern, and can be constructed by sequences of abstract machine operations.

For this pattern, we introduce higher-order abstract machine operations. That is, abstract machine operations which take sequences of abstract machine operations as parameters. These parameters correspond to the contained fragments that follow the combination of operations pattern. The example above is captured by the following abstract machine operation:

```
change_clock(save_clk: instructions,
             restore_clk: instructions,
             set_clk: instructions)
```

Conclusion. Using these patterns with existing examples, we were able to define an abstract machine that could express the behavior of any particular device driver. Although they were typically easy to recognize, it is important to realize that it was necessary to abstract from certain details in order to see the different patterns. E.g., in our experiment, the examples were mostly written by different people, who had different styles of programming, and sometimes took different approaches to the same problem. In this situation, it was necessary to determine if the same functionality could be implemented with a common structure, which happened to always be the case.

[2]Video cards have programmable clocks which can be set to different frequencies to control the video refresh rate.

4.2 The GAL Language

In this section we present the Graphics Adaptor Language (GAL) for video device driver specification. In order to understand where the language comes from, it is important to know what the essential variations are among video adaptors. The remainder of the section describes the variations that exist between cards and the corresponding constructs in GAL that capture them. A complete example of a GAL specification is described in Appendix A.

4.2.1 Ports, Registers, Fields and Params

A video adaptor is controlled by setting certain parameters stored in hardware registers of the card. These registers have addresses. A single parameter may be stored in multiple registers and only certain bits of the registers may be used. Thus the layout of the parameters on the register space is the first major variation between cards.

Access to the registers are provided through various communication schemes. As mentioned in the previous section, there is a small number of different schemes that can be used to communicate with a hardware device from a program. The choice of communication scheme is the second major variation between cards. We define several concepts to describe these notions of communication and register layout.

Ports. The first concept is the *port* which is used to define a point of communication. For example, the declaration

```
port svga indexed:=0x3d4;
```

defines a port named svga, which uses an indexed communication scheme at the I/O address 0x3d4. This is a standard port used by many video cards.

Registers. A second concept is provided by the *register* declaration, which defines how to access

Standard field	**Purpose**
HTotal, HEndDisplay, HStartBlank, HEndBlank	Horizontal resolution settings.
VTotal, VEndDisplay, VStartBlank, VEndBlank	Vertical resolution settings.
LogicalWidth	Width of virtual screen.
StartAddress	Display start address.
ClockSelect	Clock selection.
Standard param	**Purpose**
RamSize	Frame buffer memory size.
LinearBase	Address of linear space.
LinearAperture	Size of linear space.
NoClocks	Number of fixed clocks.

Table 1: Predefined fields and params.

registers on the card using the defined ports. For example, the declaration

```
register ChipID:=svga(0x30);
```

defines a register `ChipID`, which is accessed through port `svga`, at index 0x30.

Fields. The next concept is specified with a field declaration. The *field* declaration defines where a logical value is stored (in which bits of what registers) and a mapping from logical values to actual stored values. For example, the declaration

```
field LogicalWidth:=
  Control2[5..4] # Offset scaled 8;
```

defines a field `LogicalWidth`, which is stored in bits 5 and 4 of the `Control2` register and the entire `Offset` register. Additionally, the mapping clause (`scaled 8`) specifies that the value stored in the register is $\frac{1}{8}^{th}$ the actual value. The mapping is needed because cards often store a value which is some function of the field's actual value.

Parameters. Related to the field declaration, the *parameter* declaration is the definition of a constant value that is either explicit in the specification or read from the card during configuration. An example of the former case would be

```
param NoClocks:=4;
```

The majority of a GAL specification consists of the definition of fields for standard values that are used to control the video adaptors and parameters which determine certain features of the card (e.g., size of the frame buffer). Table 1 lists some of these predefined field and parameter names that can be defined in GAL specifications.

4.2.2 Clocks

A third major variation between different adaptors is the use of clocks. All adaptors have a clock which controls the frequency at which data is sent to the display. This frequency needs to be changed for different resolutions, and there are two approaches to doing this. One is to have a fixed number of frequencies to choose from, and the other is to have a programmable chip that can generate many frequencies by changing its parameters. The cards with a fixed number of clocks vary in the number of clocks and the frequencies provided, while the cards with a programmable clock vary in how the clock is programmed and its range of frequencies.

A card that has fixed clocks can be specified by defining a parameter `NoClocks` and a field `ClockSelect`. The `NoClocks` constant defines the number of clocks available, and the `ClockSelect` field defines the field which selects the clock.

For cards that have programmable clocks, a special construct is defined to specify how to program the clock. For example,

```
clock f3 is 14318*f3M / (f3N1*f3N2);
```

defines a clock named `f3`, which is programmable according to the equation on the right. The equation defines the frequency generated based on programmable values, which are defined elsewhere by the three fields `f3M`, `f3N1`, and `f3N2`. Given the desired clock frequency, the device driver uses the specified equation to find values of `f3M`, `f3N1`, and `f3N2` which approximate this frequency as closely as possible.

4.2.3 Identification

The fourth major variation observed among video cards is how the card is identified. This information is required for systems which dynamically configure themselves to use whatever card is available at that time. Card identification uses a small number of predicates which test the card and follows a decision tree to decide if the card is supported by the driver and which one.[3] Thus, we define an appropriate construct for specifying this type of decision tree in GAL.

The following is an example of this identification construct.

```
identification begin
 1: writable(Segment) => (true=>step 2);
 2: Chip_id=>(1=>oti087,others=>step 3);
 3: Chip_id2=>(0=>oti037c, 2=>oti067,
                5=>oti077);
end identification;
```

This example identifies one of four models (oti037c, oti067, oti077, oti087) of cards that use an OTI graphics controller. The construct defines a series of steps numbered 1-3 to the left. At each step, the expression to the left of the arrow is evaluated and the result is compared to the list of decisions on the right. If no decision is matched on the right, then identification fails and indicates that the driver does not support the card. Possible decisions are to identify the card or proceed to another step. Step 2, for example, reads the value of the `Chip_id` register, and if the result is 1, identifies that an oti087 is present, otherwise proceeds to step 3 for further tests. The stepwise syntax reflects the way diagnostic procedures are commonly described in manuals.

4.2.4 Modes

The final major variation between cards is that many adaptors require some flags be set under certain operating conditions. These are referred to as *modes of operation* in GAL, and are defined with the mode construct. The *mode* construct is used to specify a predicate and a sequence of assignments to fields, which enable or disable the corresponding mode of operation for the video card. For example,

```
mode HighRes:=HTotal>800;
enable HighRes sequence is Control[5]<=1;
```

This mode declaration defines a mode, `HighRes`, which indicates that a '1' must be stored in bit 5 of `Control` in order to use a video mode in which the horizontal resolution is greater than 800 pixels. In our implementation, the predicate `HTotal>800` is tested after changing the video mode; if it is true, the sequence `Control[5]<=1` is executed.

In addition to user defined modes, there are also a few built-in modes. The built-in modes have fixed predicates, but allow the specification of enabling and disabling sequences. For example, the built-in mode `SVGAMode` is true for all graphics modes and thus the user-defined enabling sequence is executed each time the mode is changed.

[3]One device driver often supports multiple cards from the same vendor.

4.2.5 Run-time variations

In addition to the variations that exist between cards, there are variations within a single driver that depend on conditions not known until run-time (of the driver). For example, some video adaptors operate differently depending on the hardware bus utilized (AT, PCI, or VLB). Additionally, if one wants to build a single device driver for a number of models from the same vendor, the variation between those models has to be chosen at run-time. In GAL, the `cases` construct is used to describe this type of variation.

As an example, the following statement is used to define the clocks for different models of S3 cards.

```
cases
for S3_TRIO32,S3_TRIO64
  field ClockSelect:=Miscr[3..2];
for others
  field ClockSelect:=Control[3..0];
end;
```

This example specifies that if the card identified at run-time is a S3_TRIO32 or S3_TRIO64, then the card has four fixed clocks selected by bits 3 and 2 of the `Miscr` field. All other cards have sixteen clocks selected by bits 3 down to 0 of the `Control` field.

4.3 Design of GAL

This section discuses some of the many forces that influenced the design of GAL. The first two subsections describe two main inputs to the design process: a definition of variations in the family and knowledge about the domain. In our case, the domain knowledge came from existing documentation used by domain engineers. Other important issues are the level of abstraction, the level of restriction, readability, maintainability, etc. While the level of abstraction and the level of restriction are of particular importance for DSLs, issues like readability and maintainability apply to both DSLs and GPLs

4.3.1 Defining Variations

One of the main inputs to the design of a DSL is a description of the variations that exist among the target set of applications. The defined variations imply requirements on the DSL in order to distinguish among instances of the program family. In our case, these variations came from a study of the documentation of existing video cards. In addition to studying different cards, inspection of the existing device drivers provided a more detailed source of variations at the implementation level. For example, given that there were a small number of ways to communicate, which varied among cards, there must be some construct in GAL specifications, which would allow the selection of the correct communication scheme. Some of this information can also be extracted from the parameters of the abstract machine operations.

4.3.2 Domain knowledge

The other main input to the DSL design process is knowledge of the domain in terms of the abstract objects or concepts and terminology used in the domain. This knowledge may come from a domain expert or from existing natural language specifications, as in our experiment. This is an important input because it leads to a more abstract user-level DSL. An appropriate terminology provides a DSL that is familiar to people of the domain. The identified abstract objects that are affected by variations in the program family provide starting points for declarative constructs.

In this experiment, we looked at several English specifications of video cards to identify the concepts and terminology used within the domain. The clocks, ports and registers are examples of concepts in the domain that we identified. After identifying them, we considered what attributes of the objects were related to variations within the program family. Declarative statements were then defined to specify the values for the attributes that varied. Thus, the abstract objects identified in our experiment directly translated to declarative constructs in the DSL. Additionally, the relationship between the objects translated into a reference relationship in the DSL. For example, registers are defined by references to port definitions. This may suggest the use of an object-oriented analysis for DSL design.

4.3.3 Level of Abstraction

One of the most important goals guiding the DSL is to provide a high-level of abstraction. In particular, we wish to intentionally focus on raising the level of abstraction from the abstract machine level. In fact, it may be desirable to include information in the DSL, which is not even used for implementation, but may be used in analyses or for documentation.

As an example of abstraction, the abstract machine developed for the video device drivers includes operations for doing bitwise shifts and logical operations. However, these types of expressions do not appear in GAL because we intentionally introduced the idea of fields and parameters to eliminate the low-level procedural nature of these expressions. This also eliminates a common source of errors.

After a preliminary design of the language, the language and abstract machine are revised in an iterative way. The revision process must satisfy the correspondence constraint between the language and abstract machine: it must be feasible to provide a mapping from the language to the operations of the abstract machine as an interpreter. During this revision process the level of abstraction must also be considered. Although it is possible to move all of the functionality of the language into the abstract machine, making the mapping essentially one-to-one, there must be conscious decisions made about where to draw the line between the interpreter and the abstract machine. The primary consideration here is the separation of functionality from specification. The abstract machine should specify how applications in the family are implemented. The interpreter, on the other hand, should specify how to make the design decisions required to map a design specification (i.e., DSL program) into an implementation (i.e., abstract machine operators).

4.3.4 Level of Restriction

Another major concern is restricting the language. It is important to consider what types of analyses might be performed on specifications in the DSL in order to insure that the language is restricted enough to make the analyses feasible. For example, in the GAL language we have intentionally not introduced loops, which insures that all device drivers can be proven to terminate. Additionally, we perform other analyses to detect common errors in the specification by providing explicit information that is difficult or impossible to extract from general purpose languages. An example of this is checking that the bits of each register belong at most to one field. This information could not be retrieved, in general, from a driver implemented in a language such as C.

4.3.5 GPL principles

In addition to the design goals that are specific to DSLs, there are several principles of general purpose language design that also apply to DSL design. General purpose languages can also help DSL design by providing a standard set of constructs that may be restricted for use in the DSL, but would still be recognized as a common construct.

On the other hand, the cases construct introduced in GAL is an interesting example of a construct which possibly has applications in DSLs in general (when a predefined abstraction may, conditionally, have one of several definitions), but is not useful for GPLs, since the behavior is totally described by the program itself and abstractions are explicitly invoked. One of the main purposes of introducing a DSL and an application generator is to embed knowledge about how to implement certain operations of the domain into the application generator. As a result, there are often declarative constructs in DSLs that are translated into executable code by the application generator, which is not generally true of general purpose languages. Since these declarations really imply operations, there is often a need to make choices between the implied operations that can only be made at run-time. This leads to the type of dynamic selection of multiple definitions that is provided by the cases statement. Since a main motivation of utilizing a DSL is to raise the level of abstraction, it will be common for DSLs to have declarative objects which imply operations and require this dynamic selection. Thus, we suspect that this construct will be useful in DSLs in general, and in fact have found it necessary in other DSLs that we have experimented with. This suggests that there are new constructs and principles that are interesting and unique to DSLs and warrant study.

5 Results

In this section we present the results of applying our framework to the domain of video device drivers. The results are presented in terms of the advantages we have gained from using our approach for this family of drivers. There are two aspects of the approach that led to these advantages. One aspect is the use of DSLs and application generators in general, and the second is specific to our framework for application generator design.

5.1 Domain Specific Language

The GAL language demonstrates many advantages of using an application generator with a DSL for the video device driver domain. These benefits include an increased level of abstraction, the possibility of automated program analyses, reuse, and productivity.

```
Profile of S3_TRI064 + S3_TRI032
Maximum resolution: 4088x2047
Maximum virtual screen: 3328x2520
        (with maximum RAM)
Ram size: 512k-8192k
Clock range: 135-270MHz
Resolution limited to 2304x1728 by
        the clock (max. refresh 67Hz).
```

Figure 2: An extract of generated S3 card profile.

There are two significant examples of the benefit of a higher level of abstraction. The first, already discussed in section 4.3.3, is the use of ports, registers, and fields to abstract from the low-level bitwise operations that would otherwise have to be used. This eliminates many common errors, is more readable, and easier to write. A second example is an abstraction from implementation. The X Window server can be considered a framework, where the device driver provides the additional functions. As with any framework, the device driver needs to be implemented in a certain way in order to be compatible with the server and requires considerable knowledge about the framework. Using an application generator, knowledge about the framework and compatibility issues are coded in the application generator, and hidden from the designer.

GAL also demonstrates that automatic analyses can be performed on the DSL, which would not be possible or feasible with a general purpose language. Example analyses that are performed on GAL specifications include detecting unused definitions, checking for exhaustive identification of video cards, identifying overlap in field definitions, checking for minimum requirements on predefined fields, and generating a card profile (summary of card characteristics). None of these analyses would have been feasible on the existing device drivers implemented in C. Using GAL not only makes the analyses feasible, but also easy to implement. For example, all of these analyses for GAL were implemented within a single day.

One particularly interesting analysis is the one which generates a card profile. Generating a card profile is an analysis which, from the GAL specification, produces a summary of the video modes that are supported by the generated device driver.

Figure 2 shows an extract of the profile generated for the S3 specification listed in Appendix B. A profile is generated for each subset of cards in the specification that have the same profile. The figure shows the profile for the S3_TRIO64 and S3_TRIO32. This summary can be compared with vendor specifications to find mistakes in field definitions and provides automatic documentation of the specification.

Finally, using an application generator provides reuse by capturing design knowledge. In the domain of video device drivers there are large benefits of reuse because there is a large growing number of video cards which could potentially be generated from a single application generator. The amount of productivity gained depends on the ease of building the application generator and consequently on the approach to its design. Thus, we discuss productivity measurements in the next section with respect to our framework.

5.2 Our Framework

In addition to the advantages obtained from the DSL approach, there are several advantages demonstrated by GAL due to our framework of generator design. The experiment shows that the framework achieves automatic and predictable generation of efficient video drivers, and a high-level of reuse. GAL also demonstrates that the benefits of the two-level approach for analyses and multiple implementations are of practical value.

5.2.1 Reuse and Productivity

The abstract machine for X Window device drivers consists of 95 small C procedures totaling 1200 lines. Implementing the abstract machine has roughly the same difficulty level as implementing a single driver directly, as the code is very similar. Since we had existing device driver implementations, some of the abstract machine code could be reused from those drivers.

The interpreter for GAL consists of 4300 lines of C code and an automatically generated parser, much of which concerns building an environment and look-up routines for declarations. Thus, together the system consists of about 5500 lines of C code. We can compare this to the size of the existing hand-coded drivers which averaged about 1500 lines. Though the effort required to build an interpreter should be less than that for building a device driver, we can estimate that the application generator requires a little more than 3.5 times the effort of an individual driver (assuming code size proportional to effort).

For the version of the X Window server we used, the existing drivers together consisted of 35000 lines of code. The GAL specifications that have been written are at least a factor of 9 smaller than the corresponding existing C driver . We can then estimate that these drivers could be generated from less than 4000 lines of GAL specifications plus the 5500 lines of the generator, totaling less than 10000 lines. This is an estimated productivity gain of a factor of 3.5. In practice there would be a higher gain, since GAL specifications are easier to write then the corresponding C driver. In addition, having an interpreter for GAL provides a prototyping environment.

5.2.2 Efficiency

Here we consider two measures of efficiency: object code size and execution speed. Although designing an interpreter is easier than designing a compiler, there are significant losses in speed and size (compared to compilation). In terms of speed, interpreters are typically 10-100 times slower than compiled programs, and in terms of size, our GAL interpreter is 10 times larger than a typical driver in object code size. However, a benefit of using partial evaluation is that we can regain the loss in efficiency.

We used Tempo [8], a partial evaluator for C, as the program specializer used to translate GAL specifications to abstract machine programs, and to produce an efficient implementation of the abstract machine programs. In order to make a size comparison, we compared the object file sizes of the generated drivers to that of the hand-coded drivers. On average, the generated driver is only 30% larger than the hand-coded one.

The speed of most of the device driver functions are insignificant, as they are only called during configuration. However, we picked three device driver functions used for drawing lines and rectangles in hardware to benchmark performance. Since the interpreter level of our framework is guaranteed to be eliminated (see section 2), we are only concerned with the abstract machine layer.

For comparison, we prepared three versions of the X Window server for an S3 TRIO64V+ video card on a Pentium PRO-200. Table 2 shows the timing results for the three servers. The S3 XAA server is the X Window server provided with XFree86 and the included hand-coded S3 device driver. S3 AM

Server	lines/s	percent
S3 XAA	189,000	100
S3 AM	150,000	79
S3 PE	191,000	100
Server	**rectangles/s**	**percent**
S3 XAA	203,000	100
S3 AM	169,000	83
S3 PE	205,000	101

Table 2: Performance results.

is the same server with a device driver which directly uses the abstract machine. Finally, S3 PE is the same server using the abstract machine, but after partial evaluation. The table shows the performance of these servers for lines and filled rectangles of size 10 as measured by the standard XBench benchmark utility.[4] The table also includes a percentage using S3 XAA as a baseline.

The table indicates that there is a loss of about 20% in performance from the use of the abstract machine. This loss of performance can be contributed to error checking, interpretation, function call, and data copying overhead. Data copying is due to the need to communicate across abstract machine operations. The write operation includes error checking to insure that if previous operations fail the resulting data is not written to the card. This is particularly important because the card could otherwise be damaged. Finally, the I/O operations require some interpretation of their parameters to determine the type of I/O to perform and which addresses to use. Although directly using the abstract machine incurs this performance loss, the results for the S3 PE server show that the program transformations performed by partial evaluation are able to recapture all of the performance loss. A majority of the error checking can also be eliminated using Tempo because often the operations preceding write operations can not fail, and thus error conditions do not need to be checked. Finally, the parameters which are interpreted to select the type of I/O to perform and used for address computation are known and eliminated by Tempo. Tempo also performs inlining and copy elimination which eliminates function call and data copying overhead.

[4]A small size is used to minimize the effect of the hardware.

5.2.3 Analyses

Our framework for application generator design contributes in two ways to the use of program analyses. The generation process is predictable and can be analyzed, and the separation of the abstract machine from the interpreter allows analysis at the abstract machine level.

As an example, the GAL abstract machine includes operations that allocate and deallocate temporary storage and operations which use the temporary storage. As long as the operations which use the temporary storage are only used between a set of allocate and deallocate operations, we can insure there will be no uninitialized pointer dereferences. The analyses of partial evaluation are capable of producing a specification of all the programs that could possibly be generated by the partial evaluation process. From this, we can obtain a formal description of all possible abstract machine programs that could be generated, and can check that the operations are always generated in the correct order. Thus, for the GAL system we can prove that uninitialized pointer dereferences will never occur. This description of the generation process may also be analyzed for performance properties, for example.

The separation of the abstract machine and the DSL provides an intermediate level at which analyses can be performed and could allow analysis at run-time. In fact, this separation corresponds to a standard technique of program specification, which factors the verification process into two parts [3]. As an example of analysis at run-time, we may wish to check that device access within a video driver is safe (e.g., does not access the disk device). This cannot be done until run-time because it depends on what devices are present at run-time. In this case, we might accept video drivers in abstract machine form and analyze the abstract machine at run-time. Partial evaluation can be performed at run-time [9], so the efficiency can still be recaptured. This kind of analysis is not feasible on machine code or even Java bytecodes due to their general purpose nature. In proof-carrying code [17], the burden of proof is put on the programmer and the proof is sent with the code to be verified (verification being easier), whereas here we make the proof easier so that it can be done at run-time.

5.2.4 Multiple implementations

The video device driver family also demonstrates a useful application of having multiple implementations of interpreters and abstract machines. In this

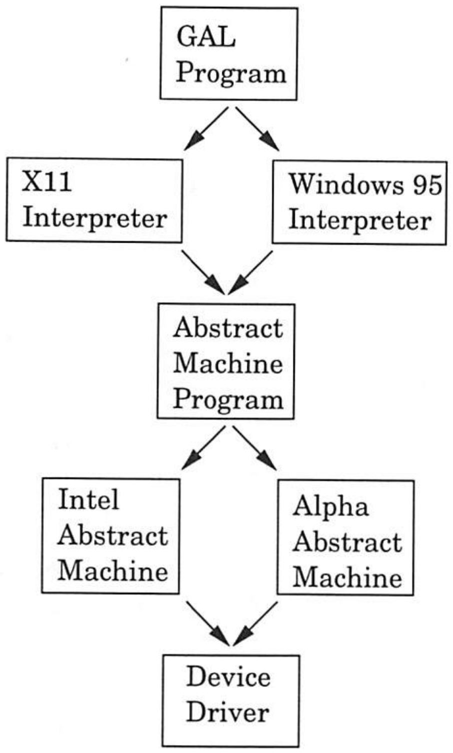


Figure 3: Multiple implementations.

domain, it would be desirable to have abstract machines for several architectures and interpreters for different operating systems. For example, Figure 3 shows the situation where there are implementations of interpreters for Microsoft Windows 95 and Linux/X11, and implementations of the abstract machine for the Dec Alpha and Intel based computers. In this situation, with the equivalent of two application generators (interpreter/abstract machine pairs), the same GAL specification can be used to generate four different device drivers. We have implemented the X11/Intel path of Figure 3.

For prototyping, we have also benefited from having a second implementation of the abstract machine which simulates the abstract machine operations. The simulation records the values that would be written to the card by the real abstract machine. This is an important feature as some video adaptors can be damaged by writing inappropriate values to the card.

6 Conclusions and Future Work

Domain specific languages hold the promise of delivering high payoffs in terms of software reuse, automatic program analysis, and software engineering. In this paper we have presented GAL, an example of a complete DSL for a realistic program family: video device drivers. We also demonstrated the benefits of DSLs by showing how GAL raises the level of abstraction of device driver specifications and identifying some analyses that can be performed on GAL specifications because it is domain specific.

A further contribution of the paper is to validate our framework of application generator design by applying it to this program family to provide an implementation of GAL. Since our implementation is based on partial evaluation, it provides a complete interpreter for prototyping device drivers, but still automatically generates efficient device drivers. Efficiency is demonstrated with results comparing hand-coded drivers to automatically generated device drivers. Generated drivers are roughly one third larger than hand-code drivers and perform equivalently in terms of speed. Additionally, we give measures on expected reuse benefits; GAL specifications are roughly a factor of 9 smaller than a driver hand-coded in C.

Although our framework significantly reduces the development time of application generators, future work could be done in this direction. Specifically, this approach would benefit from a generator-specific reuse method that would allow interpreters and abstract machines to be constructed from reused composable parts. Additionally, given the nature of DSLs, they are extended frequently to adapt to new program requirements, and the ease of extension also needs to be considered for such language components.

Our implementation of the static analyses indicates that methods of quickly constructing static analyses should also be investigated (e.g., composable analyses). This is more important for DSLs than GPLs, since static analyses are a major motivation of the approach.

In this work we have presented an application of our approach to a program family with existing family members. To further validate the approach, it is also important to study its application to a program family which is not pre-existing. In this case, the abstract machine and DSL might be developed from the results of a domain analysis or a commonality analysis, such as FAST [11].

References

[1] B.R.T. Arnold, A. van Deursen, and M. Res.

An algebraic specification of a language describing financial products. In *IEEE Workshop on Formal Methods Application in Software Engineering*, pages 6–13, April 1995.

[2] J. Bentley. Programming pearls: Little languages. *Comm. of the ACM*, pages 711–716, August 1986.

[3] H.K. Berg, W.E. Boebert, W.R. Franta, and T.G. Moher. *Formal Methods of Program Verification and Specification.* Prentice-Hall, EngleWood Cliffs, NJ, 1982.

[4] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In *Coordination and models, Proceedings of the first international conference, Cesena, Italy*, number 1061 in LNCS, pages 75–88, 1996.

[5] E. Bjarnason. Applab: a laboratory for application languages. In L. Bendix, K. Nørmark, and K Østerby, editors, *Nordic Workshop on Programming Environment Research, Aalborg.* Technical Report R-96-2019, Aalborg University, May 1996.

[6] J. Bosch and G. Hedin, editors. *Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models, Linköping.* Technical Report 96-173, Lund University, April 1996.

[7] J. Graig Cleaveland. Building application generators. *IEEE Software*, July 1988.

[8] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [10], pages 54–72.

[9] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23^{rd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996.

[10] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in LNCS, February 1996.

[11] N.K. Gupta, L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss. Auditdraw: Generating audits the fast way. In *Proceedings of the Third IEEE Symposium on Requirements Engineering*, pages 188–197, January 1997.

[12] N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[13] N.D. Jones. What *not* to do when writing an interpreter for specialisation. In Danvy et al. [10], pages 216–237.

[14] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, EngleWood Cliffs, NJ, June 1993.

[15] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th IEEE International Conference on Software Engineering ICSE-18*, pages 542–553, 1996.

[16] David Ladd and Christopher Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, New Mexico, October 1994.

[17] G. Necula. Proof-carrying code. In *Conference Record of the 24^{th} Symposium on Principles Of Programming Languages*, pages 106–116, Paris, France, January 1997. ACM Press.

[18] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. http://www.sunlabs.com/people/john.ousterhout/, 1997.

[19] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for operating system specialization. In wDSL'97 [22].

[20] Scott Thibault and Charles Consel. A framework for application generator design. In *Proceedings of the Symposium on Software Reusability*, Boston, MA, USA, May 1997.

[21] A. van Deursen and P. Klint. Little languages: little maintenance? In wDSL'97 [22].

[22] *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997. Computer Science Technical Report, University of Illinois at Urbana-Champaign.

[23] The XFree86 Project. http://www.xfree86.org/.

A A Complete GAL Example

Appendix B gives a complete listing of the GAL specification for several models of S3 video adaptors. In this appendix, we explain some of the constructs that were not included in the main text.

Although the various registers of video cards are typically accessed using an addressing scheme, there is sometimes a sequential procedure that must be followed to access some registers. The `serial` construct is used to specify this kind of procedure (see listing). The construct consists of a list of sequences of actions that should be performed on the ports to access the registers. Thus multiple ports may be accessed during the procedure, as in the example. Each sequence consists of a port, an operation (<= write, <=> read/write, => read), and a sequence of values for writes or registers names for reads and read/writes. The actions in the sequence are performed from the first port to the last, from left to right in the sequence. The mode (`R` read, `R/W` read/write, `W` write) to the right of the sequence indicates whether this sequence applies to reading the registers to writing the registers or both.

The serial construct in the example defines the registers `PLL1`, and `PLL2`. In order to write values to these registers the construct would be executed as follows. Write 3 to `misc[3..2]`, write the value of `PLL1` to `seq(0x12)`, write the value of `PLL2` to `seq(0x13)`, and finally, write 0, then 1, then 0 to `seq(0x15)[5]`.

The S3 specification also includes an example of a *derived field*, which is not discussed in the paper. This is a field whose value is derived from one of the standard fields. In the example, `StartFIFO` is a derived field. Its value is set whenever the graphics mode is set, and is based on the value of `HTotal`, the horizontal resolution. The declaration indicates this with the `from` clause.

The `clockmap` is used when a card has both fixed and programmable clocks such as the S3 Trio cards. It indicates which clocks are fixed and which are programmable. The example for the S3 indicates that clock 0 and 1 are fixed, clock 2 is not available (`NA`), and clock 3 is the programmable clock `f3`. The parameters `MinPClock` and `MaxPClock` are also related to clocks and specify the minimum and maximum values that can be generated by the clock (i.e. not all values of `f3M`, `f3N1`, and `f3N2` are valid).

Finally, the operating mode `access` is used to lock an unlock registers on the card.

B GAL S3 Listing

```
-- List all cards/models supported by this driver.
chipsets S3_911,S3_924,S3_80x,S3_928,S3_864,S3_964,S3_866,S3_868,
         S3_968,S3_TRIO32,S3_TRIO64;

-- Define ports.
port svga indexed:=0x3d4;
port seq indexed:=0x3c4;
port misc := 0x3cc, 0x3c2;

-- Define registers.
register Miscr:=misc;
register Slock:=seq(0x8);
register Offset:=svga(0x13);
register ExtChipID:=svga(0x2e);
register ChipID:=svga(0x30);
register Memory:=svga(0x31);
register State:=svga(0x36);
register Lock1:=svga(0x38);
register Lock2:=svga(0x39);
register StartFIFOr:=svga(0x3B);
register Misc1:=svga(0x3a);
register Control:=svga(0x42);
register Control2:=svga(0x51);
register HOverflow:=svga(0x5D);
register VOverflow:=svga(0x5E);
register Control3:=svga(0x69);
-- Serial registers (see appendix A).
serial begin
  misc[3..2]<=  (3,-   ,-,-,-) W;
  seq(0x12)<=>  (-,PLL1,-,-,-) R/W;
  seq(0x13)<=>  (-,PLL2,-,-,-) R/W;
  seq(0x15)[5]<=(-,-   ,0,1,0) W;
end;

-- Define predefined fields

-- Horizontal resolution fields.
field HTotal := HOverflow[0]#std;
field HEndDisplay := HOverflow[1]#std;
field HStartBlank := HOverflow[2]#std;
field HStartRetrace := HOverflow[4]#std;

-- Vertical resolution fields.
field VTotal := VOverflow[0]#std;
field VEndDisplay := VOverflow[1]#std;
field VStartBlank := VOverflow[2]#std;
field VStartRetrace := VOverflow[4]#std;

-- Virtual screen fields.
field LogicalWidth := Control2[5..4]#Offset scaled 8;
cases
for S3_928,S3_968,S3_TRIO32,S3_TRIO64
  field StartAddress := Control2[1..0]#Memory[5..4]#std;
for S3_80x
  field StartAddress := Control2[0]#Memory[5..4]#std;
for S3_864,S3_964
  field StartAddress := Control3[4..0]#std;
for others
  field StartAddress := Memory[5..4]#std;
end;

-- Define derived fields (see appendix A).
field StartFIFO from HTotal := HOverflow[6]#StartFIFOr offset 10 scaled 8;

-- Special S3 flags that must be set for 256 color graphics modes.
enable SVGAMode sequence is Misc1[4]<=1,Memory[3]<=1;

-- Define standard parameters.
param TwoBankRegisters:=false;
param InterlaceDivide := true;

cases
for S3_911,S3_924
  param RamSize:=State[5] mapped (0=>1024,1=>512);
for others
  param RamSize:=State[7..5] mapped (0=>4096,2=>3072,3=>8192,4=>2048,5=>5120,
6=>1024,7=>512);
```

```
end;

-- Define clocks.
cases
for S3_TRIO32,S3_TRIO64
  param NoClocks:=4;
  field ClockSelect:=Miscr[3..2];
  param MinPClock:=135;
  param MaxPClock:=270;
  field f3M:=PLL2[6..0] offset 2 range 1 to 127;
  field f3N1:=PLL1[4..0] offset 2 range 1 to 31;
  field f3N2:=PLL1[6..5] mapped (0=>1,1=>2,2=>4,3=>8);
  clock f3 is 14318*f3M / f3N1*f3N2;

  clockmap is (fixed,fixed,NA,f3);
for others
  param NoClocks:=16;
  field ClockSelect:=Control[3..0];
end;

-- Identification procedure.
identification begin
  1: ChipID[7..4] => (0x8=>step 2, 0x9=>S3_928, 0xA=>S3_80x, 0xB=>S3_928,
0xC=>S3_864, 0xD=>S3_964, 0xE=>step 3);
  2: ChipID[1..0] => (0x1=>S3_911,0x2=>S3_924);
  3: ExtChipID => (0x10=>S3_TRIO32, 0x11=>S3_TRIO64, 0x80=>S3_866,
0x90=>S3_868, 0xB0=>S3_968);
end;

-- Register locks on S3 chips.
enable access sequence is Lock1<=0x48,Lock2<=0xA5,Slock<=0x6;
disable access sequence is Lock1<=0x00,Lock2<=0x5A,Slock<=0x0;
```

Domain Specific Languages for *ad hoc* Distributed Applications

Matthew Fuchs
Walt Disney Imagineering
1401 Flower St., POB 25020
Glendale, CA 91221-5020
matt@wdi.disney.com

Abstract

The Internet provides a medium to combine human and computational entities together for ad hoc *co-operative transactions. To make this possible, there must be a framework allowing all parties (human or other) to communicate with each other. The current framework makes a fundamental distinction between human agents (who use HTML) and computational agents, which use CORBA or COM. We propose DSLs as a means to allow all kinds of agents to "speak the same language." In particular we adopt some ideas (and syntax) from SGML/XML, especially the strict separation of syntax and semantics, so each agent in a collaboration is capable of applying a behavioral semantics appropriate to its role (buyer, seller, editor). We develop the example of a card game, where the syntax of the language itself implies some of the semantics of the game.*

1 Introduction

The Internet is a large collection of entities, some computational, some human, each evolving independently, with its own goals, strategies, and capabilities. Our goal is to support cooperation among them, both *ad hoc* and institutionalized. This implies coordinating both human and computational agents. Popular current technologies, such as the WWW and CORBA, do not adequately support co-operation because they cannot deal with the heterogeneity among these agents and their goals. Domain specific languages (DSLs), however, can provide a framework for overcoming these difficulties

This kind of heterogeneity is a particularly difficult problem the Internet, and the Web, have not yet dealt with. The current approach, smoothing over differences through general protocols, such as the use of HTML for user interface and Java for applets, only works because of the server-centric nature of the Web. Essentially all computation occurs at the server (such as through CGI scripts) or is directed by the server (such as Java applets and even the display of HTML pages). There is no real communication between the information from the server and the client's environment beyond the browser.

When all code and all applications are developed under a central control, the heterogeneity problem doesn't exist. In our early work with mobile objects[5] we developed both display markup languages and object interfaces; as long as all the code was developed by a single source, all went well. However, our goal was to support *ad hoc* cooperation, and these approaches did not scale well across the range of agents.

If we wish to build distributed applications in a future Web, we cannot assume the reader of a Web page is a human staring at a browser. It may be an application developed entirely by the client. Yet there is no formal way to extract information from a Web page and store it in a database, or to connect forms and pages to workflows. Conversely, applets *are* objects with interfaces accessible to other objects on the client's machine, but this is only useful if their interface supports the client object's needs, as opposed to just other objects from the same server. As is the case in the database world, the server cannot know all the ways a client will want to use its information. Then there is the further problem of combining information across a variety of servers in a seamless fashion.

Domain-specific languages may hold the key to dealing with heterogeneity. As we shall see, they subsume both text markup languages and class interfaces in a powerful way. They are easily transmitted. If they are truly domain specific, they imply little about their implementation, leaving the client free to support a variety of implementations, each specific to a particular purpose.

The rest of this paper will expand on our approach of using DSLs as a means of supporting heterogenous agents in an open network. Section two will expand on the problem. Section three explains the significance of DSLs, while section four describes two short examples of DSLs in use. Section five describes the approach we have been taking. Other work is described in section six and we conclude in section seven.

2 How heterogeneity impedes distributed systems on the Web

The current World Wide Web architecture assumes the existence of three entities - the server, the browser, and the (human) user. The server sends information to the browser to be displayed to the use. Occasionally the user enters some information into the browser to be sent back to the server, which responds with some new information. If the user wants to enter the information in some local application it must be done manually. If the user wants to see two screens, or enter information from one screen into a form from another server (essentially creating an ad hoc distributed application), it must be done manually.

We want to replace "the browser and the user" with "a local agent," which may just be a browser and a user, but not necessarily. This local agent receives information – text or objects – from one or more servers and does something with it. What that something is may be to display it, store it, place it within new documents, email it, trigger some transaction, etc. But it is a local decision what to do (which might merely be executing code from the server).

We also want to enable clients to combine information from a variety of servers, or arrange for the servers to communicate with each other. This turns the current Web architecture on its head to treat the whole network in a peer-to-peer way.

Our approach to this question grew out of our experiences developing Dreme, a dialect of Scheme with mobile objects, and what appeared to be the basic asymmetry between communication with humans and communication with programs. In this scenario, suppose an object migrates to your desktop and wishes to communicate with you. If you happen to be a:

- Human, then the conventional mechanism is through a GUI. The GUI provided a sequence of pictures and simple responses for the human to interact with the object/agent. Software engineering "conventional wisdom" insists this will constitute the bulk of the object's code. Our earliest efforts looked at traditional toolkits, such as Athena, then moved to SGML-based markup languages in the quest for platform independence. HTML with forms would also seem a good choice here.
- Computer program, then the conventional mechanism is through some kind of functional interface, such as method invocation. The best candidate for this was the OMG's IDL. In conjunction with a distributed CORBA implementation, an object would be able to seemlessly communicate with other objects anywhere and bring its interface along with it.

These two approaches turn out to be very different on a number of levels. On the face of it, they are incompatible in any realistic way. Other than during program development, humans will not interface directly with objects through an IDL interface, especially naive users unfamiliar with the interface. Even more absurd would be the notion of humans communicating with each other through IDL defined interfaces. Conversely, it is highly unlikely objects will communicate with each other using HTML forms. This is, at present, a common way to support human/object communication (through CGI scripts), but even there it is unwieldy.

In an information-rich, networked universe, supporting both mechanisms is very onerous. For an object to communicate with both other objects and humans, it effectively needs two user interfaces. Just as a matter of limited resources, it is unlikely both will be complete, and there is also the problem of possible inconsistencies between them. Requiring two UIs is an undesireable feature from an engineering point of view.

The sender of an object may have a clear idea of the object's goals at the recipient, but to the recipient the object is also a resource to be manipulated for the recipient's purposes. A server may send a Java applet to a client to accept information for a purchase order, but the client will also want to log the transaction locally, and perhaps have the purchase order approved by local financial systems before it is finally submitted. And the client's interest does not end there; it is important to ensure the appropriate information is transmitted back to the server. Because the interface between the local applet and server is essentially private, and because a malicious applet can internally alter itself so the malicious code is garbage collected, the client has good reason to know exactly what purchase information passed to the server[4].

3 DSLs for distributed applications

We consider domain specific languages as the means to resolve the heterogeneity issue. In this paradigm, an object's interfaces are the languages in which it expresses the information it carries and its processing requirements.These domain specific languages are normally small, often much smaller than HTML, a popular domain-specific language for displaying simple hypertext. Whether a particular language is Turing-complete will depend completely on the particular language, but it is more likely the implementation language will be than the language itself.
In our approach we separate language semantics into two levels:

- The *abstract* semantics, corresponding to the objects in the domain itself, without any regard for actual implementation. This corresponds to what is generally considered as semantics.
- the *operational*, or *concrete*, semantics, corresponding to however the recipient processes that message. We will discuss how this processing might be done, and how we have done it, but this does not presume that all message recipients will do likewise. This looseness is one of the strong points of our approach, as it allows a variety of different recipients to be swapped in and out of a distributed application.

In the "usual" programming language, the abstract semantics is usually straightforward (the syntax was created to reflect the semantics) and the operational semantics is simply the implementation of the abstract semantics on a particular machine.
In the world of DSLs, we can still consider the operational semantics to implement the abstract semantics on a machine, but we must enlarge somewhat our concept of machine. Mawl[9], for example, is implemented on an HTML/HTTP "machine." If we generally consider the receiving entity as a machine, our view of "machine" becomes quite broad. The DSL implementer may view a corporation as a machine with workflows, email to be sent, database entries to read, store, or update, etc., if that is the domain. Or the domain may be far smaller, like a card game. Any particular site might actually have many machines defined for the different ways they might want to handle the incoming information.
Our approach is a social model. Exchanged strings represent actions by the various parties. The closest analogy is with speech acts, as first discussed by Austin[1] and then extended by Searle[12]. Traditional logic has always considered sentences to be statements about the state of the world. As such they could be assigned truth values based on their correspondance with actual fact. Austin was the first to notice that some statements were actually acts themselves, and their utterance changes the state of the world. Austin called these speech acts. The classic example of a speach act is the marriage ceremony. At the end of the ceremony, both parties say "I do." These two utterances initiate the marriage, so they have, in fact, changed the state of the world.
According to Searle, for a speach act to be successful, it not only needs the right syntax, it also needs the appropriate context. The marriage speech acts, at least in the United States, can only be performed by unmarried individuals in the presence of an authorized individual, such as a justice of the peace. If performed by actors during a show, then it is not a valid as a marriage creating speech act. It is still a speech act, but one of a different kind.
We can take a similar approach using model theory. Although there may be a general domain model, each message is interpreted using a local model (and possibly more than one, if there are multiple operations to be performed). The net effect of interpreting a message using the local model, however, must be congruent with the abstract model. Accepting a purchase order should eventually lead to a shipment and a bill.
Model theory also leads, in a roundabout way, to justifying the application of DSLs. Just as there may be many models for a particular language, there may be many languages to express a particular model. For each of these languages there is an interpretation function to describe the mapping between it and the model. The interpretation functions for programming languages usually appear relatively straightforward. They describe how statements in the language function in general. However the interpretation function from a program in a traditional programming language to a particular idiosyncratic domain will be far more complex. Given that the interpretation function can vary in complexity, we would argue the appropriate language for any domain is the one with the simplest interpretation function. This is certainly the case where the language syntax corresponds directly with domain semantics so there is a one-to-one correspondance between language elements and the corresponding domains.
We've sidestepped the issue of implementing the language, or even of the variety of implementations corresponding to the different operational semantics we might require. But we've subdivided the initial

problem – how to implement a variety of apparently unrelated applications in a particular problem domain – into two smaller problems:

- What is the language that best expresses the semantics of the domain.
- How can we implement this language to express the various operational semantics we need.

We maintain this is simpler. The first item is mostly a design issue, which needs to be addressed in any case. For the second one, each operational semantics corresponds roughly to an application, but it is now a mapping from a particular language to a particular implementation. Most of the operational semantics will fall into a small group, such as GUI, storage and retrieval, etc. Since we have subsumed the different problem domains under a particular structure (a language) and have now factored the different ways we'll need to manipulate those structures, we can attack each area separately (how to display a message, how to decompose and store a message, etc.). Our prefered method would be to use metalanguages to describe these mappings.
We can look at objects in message-passing based languagas, such as C++ and Java, as implicitly being language processors as well. In these languages, each object has one or more interfaces it presents to the rest of the world (C++ objects have one based on its class, Java objects can have several). Each interface describes a set of messages, called methods, accepted by the object. We can consider the set of methods as the alphabet of the object's language. After creation, an object receives a (potentially infinite) list of method invocations. Methods not in the alphabet are gibberish. (Dynamic languages, like SmallTalk, may have a default means to handle this; statically typed ones catch these at the compilation stage.) Otherwise the list of messages form a string in the language defined by the interface. Interface definition languages for object oriented languages currently specify no more that the alphabet, so any string (sequence of method invocations) is declared valid even though objects do not necessarily accept all sequences.

4 SGML/XML as DSL metagrammar

We have relied heavily on SGML, the Standard Generalized Markup Language, as the metagrammar for defining our various DSLs. SGML has some important characteristics which make it a candidate for the role:

```
<!ELEMENT element-name ((elem1, elem2)+ |
                        (elem3, elem4)*)>
```

Figure 1: Element definition

- It is an existing international standard already used to mark up terabytes of information, much of which may be interesting for the kinds of applications under consideration.
- Although rather complex, a number of parsers are available. XML, a simplified version of SGML designed for Web delivery is designed to be simple to parse.
- It is LL(1), as we will discuss later.
- Most important, SGML was designed to enable a complete break between syntax and semantics through its promulgation of logical, or descriptive, markup.
- SGML is also the metagrammar in which HTML is defined, so it will look familiar to people who have read Web document sources.

In descriptive markup, a tag designates what it is, not how it should be shown. For example, the `<date>` tag in the fragment `<date>10/29/1999</date>` indicates that the string is to be interpreted as a date not, for example, a part number. How it is to be displayed must be designated elsewhere, either in a display application or a stylesheet. However different applications - for database storage, for display, etc. - can all use the presence of the `<date>` tag.
SGML/XML does not use Backus-Nauer forms for defining grammars. An SGML rule is called an element definition. An example is given in figure 1. It has three parts:

1. The `ELEMENT` keyword to indicate this is an element definition.
2. The element name (the left hand side of a BNF production).
3. The *content model*, a regular expression what the internal contents of the element are. Where the element contains text, this is designted by the `#PCDATA` keyword.

We will give an example of a full grammar when explaining the Bridge application.

```
<!ELEMENT bridge (player+, deal,
                 bidding, dummy, play)>
<!ELEMENT player #EMPTY>
<!ATTLIST player position (north | south |
                           east  | west)
                                 #REQUIRED
                 name     cdata  #required>
<!ELEMENT deal (card+)>
<!ELEMENT card #empty>
<!ATTLIST card suit (spades | hearts |
                     diamonds | clubs)
                                 #REQUIRED
               face cdata #REQUIRED>
<!ELEMENT bidding (bid | pass)>
<!ELEMENT bid #empty>
<!ATTLIST bid  suit (spades | hearts |
                     diamonds | clubs |
                     no-trump) #REQUIRED
               tricks cdata #REQUIRED>
<!ELEMENT pass #empty>
<!ELEMENT dummy (card+)>
<!ELEMENT play (trick+)>
<!ELEMENT trick (card, card, card, card)>
```

Figure 2: Grammar for Bridge Game

5 An example – the game of Bridge

A straightforward distributed application we have tried is a Bridge tournament. Bridge is interesting because it has both an interactive component and a multilayered architecture.
Bridge is the most popular of a number of games all based on the same basic procedure. In all of them:

1. Dealer deals the cards.
2. The parties bid to determine a *trump* suit and who will play the first card.
3. The "dummy," the partner of the highest bidder, lays his cards on the table.
4. The cards are played in a series of rounds. Each round is started by the winner of the previous round.
5. The play is scored.

Other families of card games are a variation on this, such as poker, where players can exchange some of their cards before bidding.
We can write a grammar to cover the context free aspects of this schema, as in figure 2. Some aspects of a game are necessarily context sensitive (such as not playing the same card twice) – these are handled

```
<bridge>
<player position = ''north''
        name = ''author''> ...
<deal><card suit = ''spades''
             face = ''king''> ... </deal>
<bidding><bid suit = ''hearts''
             tricks = ''2''> ... <pass></bidding>
<dummy><card suit = ''clubs''
              face = ''2''>...</dummy>
<play><trick><card suit = ''hears''
                    face = ''6''>...</trick>
      ...
</play>
</bridge>
```

Figure 3: Schematic Brige Game

by the agents, who or whatever they may be. It is also possible to create another meta-language to specify some of these context sensitivities (such as most steps proceed in a round robin fashion, or play is selection without replacement).
A correct and complete game of bridge 3 is a string in this language, although (due to context sensitivity) not all strings are correct games. In a distributed game of bridge:

1. The actions of all the parties will, jointly, create a correct string in the language.
2. Each party will receive (and interpret) a correct bridge string. We can guarantee this because of the nature of bridge. There are other domains where only one party can determine that a correct string has been produced. There may also be domains where no party actually sees the final string (it may be distributed), but there is enough information to verify it is correct.
3. A knowledgeable party can view the resulting string and determine that it was, indeed, a correct game.

When playing bridge, the dealer generates the first part of the game by passing out the cards. Each player receives an open game start tag and then their 13 cards. If the entire game is automated, these may come in one message. On the other hand, if the dealer is human, the cards will be dealt one at a time, so the 13 cards will come in 13 messages. After this comes the bidding. Bidding terminates when three consecutive players pass. Each player, in turn, sends its bids to all the others, ensuring each player sees the same (correct) sequence. As the bids must escalate, this either requires some checking from the agent, or the grammar will drastically increase in size (since the number of possible games is actually finite, we could use a finite state machine, but this would not be terribly practical). It is easy enough to check the resulting string. Next comes the laying out of the dummy's hand – another 13 cards. These are sent by the dummy to the all the players (it also sends to itself for completeness). Finally the actual play begins. With each round, the leading player starts the round and plays its card, followed by the others. Again these are sent to all the players until the game terminates. At the end of the game, each player will have received a correct string. They will be the same except for the initial cards dealt.

As long as the players generate their pieces of the string correctly, it does not matter if they are computational objects or human. Non-participants or judges can listen in and follow that part of the game open to the public. The string can also be stored, printed, spoken, compared with other games, or otherwise manipulated.

If we want to compare computational and human agents, we can see that support in either direction is incremental. In the former case, the string only needs to be parsed into a data structure. Beyond that is the need to produce agents capable of playing bridge at whatever level is desired. This is outside the scope of this paper. In the human case, the interface can be as simple or complex as desired. A human could function with a simple, command line interface for Bridge, reading the messages directly and typing in bids and cards. This approach rapidly reaches a point of diminishing returns as the information becomes more complex, so a more sophisticated user interface is required.

Because the language is public, any client understanding the language is a valid participant. Whoever defines the Bridge language has created a public protocol. Bridge players can build their own clients or retrieve them from anywhere.

Maintaining the string representation down to the lowest level can facilitate *ad hoc* collaboration. Suppose, while playing bridge, I retrieve a bridge advisor program. Somehow I need to communicate the current state of the game to the advisor, and it needs to be able to give me feedback. If a traditional OO approach is taken, access to the GUI is encapsulated in the bridge client through a tangle of widgets and callbacks. Either the bridge client has a separate interface for other computational agents (such as a set of methods) or the information must be entered by hand. The second case is laborious and labor prone. The first, as we mentioned above, requires the client to have two interfaces - the GUI and the method. On the other hand, if the GUI, or some component of it, is really just a mapping from the current string to widgets, the advisor can retrieve the string directly from the GUI without requiring any communication with the bridge client. Ideally, the advisor can even borrow the display mapping to show the user alternative scenarios. Since the callback of playing a card is to send some tokens, it would even be possible to develop higher level tools to redirect the user's choice through the advisor (particularly if the advisor is also a tutor).

While the game string itself resulting from several processes interacting, that same string can be both program and datastructure to other applications. For example it is a program to a bridge game pretty-printer or DBMS storage routine. It is a data structure to any query facility trying to analyze the game

6 Implementation architecture

In discussing implementation, we will separate architectural issues into two parts, language considerations and application considerations.

We can also distinguish two broad application types - one in which processing is done continuous with the transmission of tokens (the bridge game) and one in which the entire language string is received by the client before any processing needs to be done (such as a purchase order).

6.1 The importance of top-down parsing

The flow of the bridge application – each player adding its piece to the construction of the final string – requires the use of an LL(1) grammar and parser for the language, as opposed to the more common LR(1) or LALR(1) grammars. These latter grammars lead to bottom up parses, while LL(1) grammars have top-down parses. In the LL parse, the parser knows which production it is entering as each token is encountered. This is imperative for the game, since the client always needs to know which state it is in to perform correctly. The LR parser, in contrast, knows which production it is leaving when the last token for the production has been seen. In other words, the parse will report to the application what *has* happened, not what *will* happen.

LL(k) parsing, in general, has been deprecated until recently because the need for a variable amount of lookahead. However, [11] has shown how to minimize the lookahead. As most rules seem to be LL(1), this makes LL(k) parsing very attractive.

An LR grammar, by contrast, appears able to support the purchase order application since the entire message should be available at the client before parsing begins. However this is not necessarily true; it might be that the document as sent contains references to other information which could be expanded in place if necessary. In other words, the sender leaves it to the receiver to determine if the additional information is necessary. For example, only a part of an item's record might be sent. If the parse is top down, the application will have more information about the message when it reaches the reference than if the parse is bottom up (this does not mean there will necessarily be sufficient information - it may be the information is necessary for something later in the document). However in a top-down parse, the additional information can be retrieved at the point the reference is encountered. In a bottom-up parse it is more likely that part of the parse would need to be discarded when it is time to retrieve the additional information. I would also argue that queries against a message are also easier in the LL scenario for the same reason. The more elaborate the path on a query, the more it resembles a parse in which most of the document is discarded. An LL grammar makes it more likely there will be a reasonable path through the document to the desired information.

6.2 Interpreter structure

The application architecture we have used is heavily influenced by our choice of Scheme for implementing our initial mobile object language. Lisp dialects are generally LL(1) because of the use of S-expressions. The feature facilitates the development of the various Lisp macro systems. A macro, in essence, is a function whose parameters are not evaluated. The body of the macro can rewrite the parameters (which may be a large chunk of code) and then have the rewritten code evaluated in place of the original code. This implies a top down evaluation. The other interesting aspect of Scheme is its support for nested lexical scopes (closures) and first class functions. By treating the parser itself as a coprocess, we have been able to write top down interpreters with several levels of nesting.

We essentially use an event-driven model, with events based on the generic identifiers of the the tags. Event handlers are grouped in lexically scoped groups. Each handler has three sections:

1. A pretraversal section for processing related to the current node in the parse tree. For example, suppose we had a purchase order and needed to keep track of the total cost for all the included items. Because of the lexical scoping, the code can simply define a variable here which will be accessed while the subtree is traversed. Also, because of the scoping, there is a possibility to use the same events recursively, so we can track prices of lists within lists.

2. A traversal section, in which the tree rooted at the current node is traversed. It is possible at this point to designate a new list of event handlers for nodes in the subtree. These event handlers are only required for events whose processing changes in the context of the current node. Since the event handlers are nested, if a handler is not found in the current scope, outer scopes are searched until one is located. In particular, as these handlers are also lexically scoped within the pretraversal section, they can have access to variables created at that level. So, for the purchase order example, the variable defined in the pretraversal state is incremented here.

3. A post traversal section for final processing after the subtree has been traversed. In the purchase order example, we would now have the total amount of the items and can print, transmit, store, etc., that value as we wish.

An example of this structure is given in figure 4. Note that due to the lexical scoping it is possible to have one event for a card when it appears during the `deal`, and another when it is part of `play`.

7 Related Work

Another approach to communicating among agents, either contrasting or complimentary depending on implementation, is given by KIF/KQML[7, 6]. This effort has grown out of the AI segment of the agent community. KIF is a predicate calculus based language for encoding ontologies – exhaustive analyses of the information in a particular domain. When used as a communication language, small Lisp-like programs are sent and executed remotely. KQML is a protocol for wrapping inter agent messages based on speech act theory. KIF seems to be very complimentary with our approach when considered as a domain specification language. The model must be described one way or another. However, while

```
('bridge
   (let ((var1 1)
 ...)
      (startup code)
      (event-list
        'deal (lambda (event)
                (let ((...))
                  ...
                  (event-list
                    'card
                      (lambda (event)
                        ...)
                    ...)
                  ...))
        ...
        'play (lambda (event)
                (let ((...))
                  ...
                  (event-list
                    'card
                      (lambda (event)
                        ...))
                  ...))
    (post-processing...)))
```

Figure 4: Interpreter Structure

the predicate calculus may be the best language for describing a domain, it is not necessarily the best syntax for making statements in that domain. The ideal DSL is the one with the most economical mapping between the syntax and semantics. The KIF community is concerned with producing ontologies - exhaustive analyses of domains. Our approach does not require that much overhead. KQML is a protocol for wrapping messages among agents heavily influenced by speech acts. Although it was developed with KIF in mind, it is agnostic concerning the language of the message and so could work with DSLs as well.

We see this approach as very compatible with *aspect-oriented programming*[8] and jargons[10]. Both of these look at decomposing a problem into a number of different aspects (or jargons), each with its own domain and language. A complete application is built by composing the program from statements in the different domain specific languages. The AOP effort weaves these together from separate programs, while the jargon approach allows the programmer to mix them in the source code. Both of these approaches are very implementation oriented. They represent possible alternative means of implementing local processing of a DSL message. The automatic weaving element of AOP is very attractive, as aspects could represent additional information the client could get from the server about the message without being completely dependent on the server for all aspects of processing.

The Shopbot[3] takes an explicitly AI view to integrating Web pages into a particular application. Shopbot is a shopping agent for the World Wide Web. Using a set of heuristics, it can learn how a shopping site is organized and help a user find products in a specific domain. This is a valid approach where:

- The problem domain is well structured but the messages are not.
- The difficulty of creating the agent is compensated by the number of uses.

Where the messags are structured in a domain appropriate way, the Shopbot approach is unnecessary. However, if it can convert unstructured messages to strucured ones, it could be integrated into such a system.

8 Conclusions

We have shown how DSLs can play an important role as the glue in multi-organizational distributed

applications in the Internet. With the strong break between language definition and implementation semantics, we can map these languages into GUIs for human agents and functional interfaces for objects, so this approach subsumes both HTML and IDL and presents a unified communication paradigm.
This approach has many similarities to EDI[2]. In EDI the messages are standardized, so they could be considered a DSL, and each party is free to process them any way they require, so long as it is congruent with the abstract semantics, as determined by the international standards bodies. However, implementing EDI has been extremely difficult, even for large corporations. We suspect that our approach could simplify EDI implementation considerably.
We intend to apply this approach in a number of problem domains. The appearance and ready acceptance of XML indicates the Web requires this kind of approach.

References

[1] J. L. Austin. *How to Do Things with Words.* Oxford University Press, 1962.

[2] Edward Cannon. *EDI Guide: a step by step approach.* Van Nostrand Reinhold, 1993.

[3] Robert Doorenbos et al. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents.* ACM, 1997.

[4] Matthew Fuchs. Beyond the write-only web. Technical report, 1995. http://cs.nyu.edu/phd_students/fuchs/in-long.ps.

[5] Matthew Fuchs. *Dreme: for Life in the Net.* PhD thesis, New York University, 1995.

[6] KQML Advisory Group. An overview of kqml: A knowledge query and manipulation language. Technical report, University of Maryland, 1992.

[7] Thomas Gruber. A translation approach to portable ontology specifications. Technical Report KSL 92-71, Knowledge Systems Laboratory, 1993.

[8] Gregor Kiczales et al. Aspect oriented programming. In *Proceedings of DSL '97.* University of Illinois Computer Science, 1997. http://www-sal.cs.uiuc.edu/ kamin/dsl.

[9] David A. Ladd and J. Christopher Ramming. Programming the web: An application-oriented language for hypermedia service programming. Technical report, 1997. http://www.bell-labs.com/project/MAWL/papers/Overview.html.

[10] Lloyd Nakatani and Mark Jones. Jargons and infocentrism. In *Proceedings of DSL '97.* University of Illinois Computer Science, 1997. http://www-sal.cs.uiuc.edu/ kamin/dsl.

[11] Terence Parr. *Obtaining Practical Variants of* LL(k) *and* LR(k) *for* $k > 1$ *by Splitting the Atomic* k-*Tuple.* PhD thesis, Purdue University, 1993.

[12] John Searle. *Speech Acts.* Cambridge University Press, 1969.

Experience with a Domain Specific Language for Form-based Services

David Atkins, Thomas Ball[*], Michael Benedikt,
Glenn Bruns, Kenneth Cox, Peter Mataga, Kenneth Rehor
Software Production Research Department
Bell Laboratories, Lucent Technologies
`http://www.bell-labs.com/projects/MAWL/`

Abstract

A form-based service is one in which the flow of data between service and user is described by a sequence of query/response interactions, or forms. A form provides a user interface that presents service data to the user, collects information from a user and returns it to the service.

Mawl is a domain-specific language for programming form-based services in a device-independent manner. We describe our experience with mawl's form *abstraction, which is the means for separating application logic and user interface description, and show how this simple abstraction addresses six issues in service creation, analysis, and maintenance: compile-time guarantees, implementation flexibility, rapid prototyping, support for multiple devices, composition of services, and usage analysis.*

1 Introduction

A form-based service is one in which the flow of data between service and user is described by a sequence of query/response interactions, or forms. A form provides a user interface that presents service data to the user (such as the time of day), collects information from a user (such as her name), and returns it to the service.

Both traditional interactive voice response (IVR) services and newer web services fit the form-based service paradigm. An IVR service typically presents a user with a menu of choices ("For Jazz Music, press 1; for Classical Music, press 2; ..."), collects a sequence of digits or performs automatic speech recognition, and then presents information or another menu. A web service sends an HTML page to a user's (graphical) browser, providing information and a set of input fields to request information such as account and password.

Many services and devices fit the form-based interaction paradigm. For example, a banking service might be described by a set of forms, independent of a particular device, such as an automated teller machine, web browser, or telephone. Presentation and collection of information will differ radically, as suited to the device. A presentation of account information to a web browser might show a table of account information, including status, balance, and interest. The corresponding presentation to a telephone would provide this information, or perhaps a subset thereof, in a conversational manner by "reading" the account information to the user. Nonetheless, the basic interaction (present account information to the user) can be specified via a generic form.

Mawl is a domain-specific language (DSL) for programming form-based services in a device-independent manner [LR95, Aea97]. Mawl separates the specification of application control flow and state management from the specification of a user interface. As a result, one can code an application that is accessible via a web browser and, with minor modifications to only the user interface specification, make the application accessible via interactive voice response (IVR) platforms.

This paper describes our experience with mawl's *form* abstraction, which is the means for separating application logic and user interface descriptions. The initial impetus for the form abstraction was to simplify the creation and maintenance of dynamic web services[1] based on the Common Gateway Inter-

[*]Correspondence contact: tball@research.bell-labs.com, Room 1G-359, 1000 E. Warrenville Rd., Naperville, IL 60566.

[1]In this paper, we will generally use the word "service" to refer to an "application", and the word "logic" to refer to

face (CGI). In particular, mawl was constructed to provide certain compile-time guarantees about the behavior of web services, as well as platform independence and implementation flexibility by hiding the details of the CGI and HTTP (HyperText Transmission Protocol [BL95]) protocols from the programmer via language abstractions.

Our continued experience with mawl's form abstraction illuminates how a simple language abstraction can improve many parts of the software development life cycle. In addition to addressing the above two problems, the form abstraction has provided straightforward solutions to four other problems:

- *Prototyping services.* Many programmers equate "prototyping" languages with typeless, interpreted languages such as perl [WS90], tcl [Ous94] or ksh [Kor86]. In contrast, mawl supports prototyping of web services via a static type system, allowing services to be tested from a web browser without the need to write any HyperText Markup Language (HTML) documents [BLC95].
- *Supporting multiple devices.* One measure of a domain specific language is how well it supports changes that were not directly planned for in its initial design. Mawl first targeted the graphical web browser, but naturally accommodates the integration of new devices into existing services. We report on our experience in developing a number of services accessible via both the graphical web browser and the telephone.
- *Composing web services.* Many web services query, collect, and integrate information from other web services (i.e., see MetaCrawler [SE95]). In most cases, the only programming interface to remote web services is via an HTTP request, which returns HTML that must be parsed to extract the desired information. The mawl form abstraction substantially simplifies the programming of services that must interact with remote web services.
- *Usage analysis.* The creation of a service is just a small part of the software life cycle. Analysis and maintenance of a service are necessary to keep the service up-to-date and functioning properly. The form abstraction is a natural place to monitor all user/service interactions and record a log of interesting events. A java applet allows programmers of mawl services to analyze usage patterns.

the specification of control flow and state management of a service.

Section 2 present a brief history of mawl and describes the basics of the mawl service architecture and language. Section 3 shows how the mawl form abstraction has helped address six problems in service creation, analysis, and maintenance. Section 4 summarizes the paper and Section 5 tells how to obtain mawl.

2 Mawl: History, Service Architecture, and Language

This section describes the brief history of mawl, the mawl service architecture and its implementation in a domain-specific language, and, finally, some details of the language.

2.1 A Brief History

Mawl was created in early 1995 because of two major difficulties experienced in programming form-based web services via Common Gateway Interface (CGI) scripts.

First, when programming such scripts in a general purpose language such as C, perl or ksh, one sacrifices traditional compile-time guarantees about the consistency of the service logic and user interface code. For example, it is difficult to automatically determine if a service will generate correct HTML, or if the service is prepared to deal with whatever data may be submitted via a `FORM` mark in the generated HTML. Such basic questions may be very difficult, if not impossible, to answer in the context of a general purpose language. Another contributing factor to this problem is that many web services are programmed in an ad-hoc fashion, lacking even a basic service architecture. As a result, service logic and user interface description are often intermingled, as shown in Figure 1(a). This makes it nearly impossible to make any sort of compile-time guarantees.[2]

Second, adherence to the HTTP/CGI protocols places burdens of low-level implementation detail on the programmer. With the HTTP request/response

[2]This problem is not confined to web services. Most programming environments for IVR services also have a very tight coupling of service logic and user interface description.

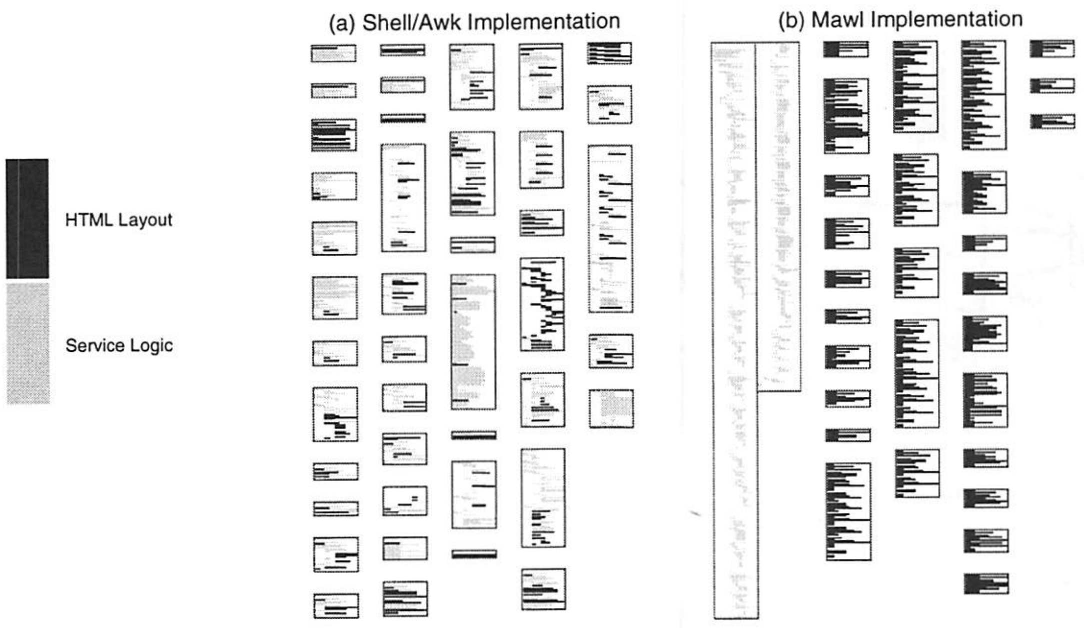


Figure 1: A global overview of two implementations of a web service called the LunchBot. Each rectangle represents a file and each line in a rectangle represents a single line of code, with length and indentation reflecting that of the underlying source code. Lines are colored to show whether they are part of service logic (yellow/grey) or user interface (blue/black). The first version of the Lunchbot (a) was programmed using shell and awk, in which service logic and HTML specification are intermixed. The second version (b) was programmed in mawl, which cleanly separates the service logic and HTML.

paradigm, a CGI process is started to respond to an HTTP request. Once the CGI process has sent the requested data, it terminates. However, many services require sequencing between pages, and the maintenance of persistent state on the server. As a result, programmers code by hand what is generated automatically by compilers for traditional sequential languages. A CGI library may provide some assistance, but the resulting program is nonetheless intertwined with a specific implementation model.

To address these problems, mawl presents an architecture for form-based service creation that is independent of the HTTP and CGI protocols. Programmers are given the illusion of a traditional imperative language in which they may code a centralized service, rather than a set of scripts coupled indirectly through HTML pages. The mawl compiler translates the program into a HTTP/CGI implementation, or into a stand-alone server implementation. As shown in Figure 1(b), mawl cleanly separates service logic and user interface (HTML layout).

2.2 The Mawl Service Architecture

Figure 2 illustrates three main abstractions in the mawl service architecture: sessions, forms, and templates. A service contains one or more *sessions*. A session specifies the control flow of a service and the update of service state (persistent and per-session), which may involve concurrency control. Typically, each session controls a different aspect of the service (e.g., there may be a session for general users and another session for the administrators of a service).

A session interacts with the user via the *form* abstraction. A form is an object that:

1) receives data from a session;
2) creates a document by dynamically parameterizing a template with the data;
3) presents the document to the user (via a browser);
4) accepts a response from the user;
5) returns the extracted user data to the session.

A *template* defines the static portion of a user interface as well as the dynamic portions that are param-

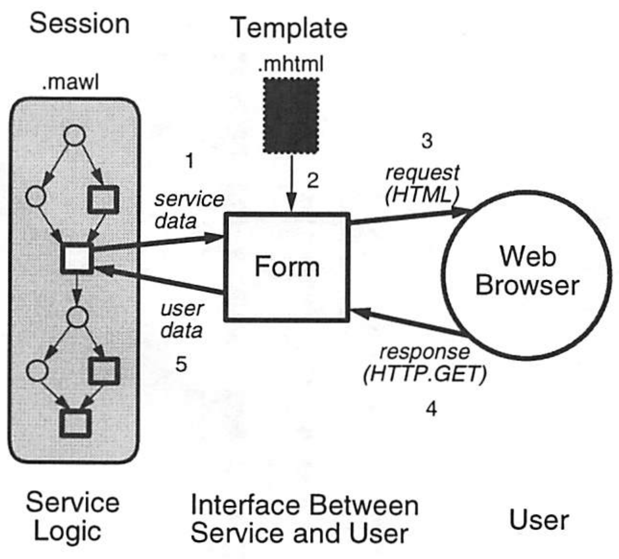


Figure 2: The mawl service architecture. Integer labels show flow of data.

eterized by values passed to the form by a session.

Figure 3 contains a simple mawl service that will be used to explain the three abstractions in more detail.

2.2.1 Sessions

A service has one or more *sessions,* each of which defines a sequence of interactions with a user. In our example, there is one session, `Greet`, that interacts twice with the user, first prompting for the user's name and then greeting the user, displaying a count of the number of visitors to the service, and the elapsed time between the presentation of the first and second pages to the user. The service logic, written in mawl, is shown in Figure 3(a). [3]

Mawl provides a persistence model that allows programmers to specify the type of storage required for mawl variables. Variables may exist on a per-session instance basis (as declared by the keyword `auto`) or persist over all session instances (as declared by the keyword `static`). In our example, the variables `GetName`, `ShowInfo`, `time_now` and `i` are per-session variables, while `access_cnt` is a persistent variable.

The only way for a session to interact with the user is through a simple input/output abstraction called a *form*, as described next.

[3]Mawl has standard imperative constructs for looping, conditional control flow, procedure calls, exceptions, etc. Mawl is a statically typed language.

2.2.2 Forms

The main role of a form is to take data (service data) from a mawl session, present it to the user, collect information from the user (user data), and return it to the session. The first two lines of session `Greet` in Figure 3(a) declare two forms, `GetName` and `ShowInfo`. A form object is declared in a session with a type signature specifying the structure of the expected service data and user data records. The form `GetName` has type `{} -> {string id}`, meaning that it expects no data from the service (thus, the empty record {}) and returns a record containing a string.

A form has a `put` method by which a service/user interaction takes place. The session provides the form's `put` method with a record containing the service data, and in return receives a record containing the user data. This is shown in Figure 3(a), where the service first provides `GetName` with its required (empty) input record and receives back a record containing the string field `id`. This string is extracted into the variable `i`. The service then supplies `ShowInfo` with a record containing three values: the user's name, the updated access count, and the elapsed time. The empty record returned by this form is ignored by the session.

There is a close connection between a form and interface definition languages (IDLs). An IDL is a language for describing the interfaces of a software component. An IDL specification describes the input/output signature of an operation, where a set of operations comprises an interface. CORBA [Gro95] (Common Object Request Broker Architecture) and RPC [Sri95] (Remote Procedure Call) both have IDL specification languages. Just as with forms, IDL programs only express the signatures of operations, but do not describe their computation. We will return to this comparison later in Section 3.

A form is associated with a template. The linkage in the example here is by common name.

2.2.3 Templates

The service data sent to a form is used to generate an interface by parameterizing a template. Templates are specified separately, in the user-interface languages appropriate to the various devices. Figure 3(b) and (c) shows templates written in the language MHTML. MHTML is an extension of HTML that is used for creating templates. In MHTML, the

```
(a) Greet.mawl:

  static int access_cnt = 0;
  session Greet {
    auto form {} -> { string id } GetName;
    auto form { string id, int cnt, int time } -> {} ShowInfo;

    auto int time_now = minutes();
    auto string i = GetName.put({}).id;
    ShowInfo.put({i, ++access_cnt, minutes()-time_now});
  }

(b) GetName.mhtml:

  <HTML><HEAD><TITLE>Get-Name Page</TITLE></HEAD>
  <BODY>Enter your name: <INPUT NAME=id> </BODY></HTML>

(c) ShowInfo.mhtml:

  <HTML><HEAD><TITLE>Show-Info Page</TITLE></HEAD>
  <BODY>Hello <MVAR NAME=id>, you are visitor number <MVAR NAME=cnt>.<BR>
        Time elapsed since first page is MVAR NAME=time> minutes.
  </BODY></HTML>
```

Figure 3: A mawl service (a) that asks the user for a name through the form `GetName` and then uses the form `ShowInfo` to display their name, how many visitors to the service there have been, and the time elapsed between the presentation of the first and second forms. The HTML templates corresponding to the forms are (b) `GetName.mhtml` and (c) `ShowInfo.mhtml`.

values of a form's service data may be accessed using the `MVAR` mark, among others. This mark indicates substitution of the value of the service data into the generated HTML. User data are represented by the standard HTML user-input marks such as `INPUT` and `SELECT`; the `NAME` attribute of these marks is the name of the user data variable.[4]

A template represents one possible "implementation" of a form (more precisely, an implementation of a form's `put` method), for a particular browser which will "execute" it. A form may have no templates associated with it, which has interesting implications for prototyping services (Section 3.3). Furthermore, a form may have multiple templates associated with it, which is useful for supporting multiple devices or browsers (Section 3.4).

[4]Note that the MHTML in Figure 3 does not contain any `FORM` mark which specifies the CGI script to be executed upon submission of the `FORM`; the mawl compiler and run-time system takes care of inserting a `FORM` mark and ensuring that control returns to the appropriate point in the session with the correct state, as discussed in Section 3.2.

2.3 Mawl Types

Mawl has four fundamental types: integers, floats, booleans, and strings; and three complex types: records, lists, and forms. Mawl records are similar to C structures. The syntax for defining a record type is to enclose a list of type specifiers and identifier pairs in braces. For example,

```
auto { string name, int age } customer;
```

defines a record named `customer` with a field `name` that is a string and a field `age` that is an integer. Record values can be constructed "on the fly", as shown in Figure 3.

Lists in mawl behave much like arrays in C, although there is no storage allocation required. A list type is denoted by enclosing another type in brackets. For example, a list of strings would be declared as:

```
auto [ string ] names;
```

The list elements are denoted using brackets and an integer index: `names[i+1] = names[i]`. Lists grow automatically to accommodate such references. List values may be formed by enclosing a comma separated list of values in brackets:

```
charlist = [ "a", "b", "c" ];
```

As illustrated earlier, the syntax of the form type specifier is the keyword form along with two type specifiers, the service type and the user type, where the types are separated by the suggestive token ->. The service and user type specifiers must be record types.

For example,

```
auto form { [ float ] temps } -> {}
               show_temps;
```

might be used to display a list of temperature values, as shown below:

```
show_temps.put( { [0,10.5,20,30] } );
```

2.4 MHTML

As explained previously, the MVAR mark in MHTML is used to insert scalar data into a template. The MITER mark is used to substitute mawl list values. A natural use of a list might be to display a table with a variable number of rows and/or columns. The construct <MITER>...</MITER> is used to iterate over mawl lists and generate HTML that is dependent on the list element values. The MITER mark uses the NAME attribute to specify the name of a list field in the form's service data. The additional attribute MCURSOR names a new cursor variable over the list. The MHTML enclosed between <MITER> and </MITER> is repeated for each element in the list, with the value of the cursor variable set to the index for that iteration. Other MHTML marks may then use mawl's list element notation to refer to list elements.

The MHTML below shows how a list of temperatures might be displayed in a single column table:

```
<HTML><BODY>
 <TABLE>
  <MITER NAME=temps MCURSOR=i>
   <TR><TD><MVAR NAME=temps[i]></TD>/TR>
  </MITER>
 </TABLE>
<BODY><HTML>
```

The <MITER> mark iterates over the temps list and i is the name chosen for the index used in the subsequent MVAR mark.

3 Experience with the Form Abstraction

This section discusses how the form abstraction helps to address six different problems in form-based services. In addition, we touch upon various languages issues that arose and discuss related work.

3.1 Compile-time Guarantees

In many web services and IVR services, service logic and user interface code are inextricably interleaved, as shown in the old version of the Lunchbot (Figure 1(a)). Consequently, reasoning about one inevitably requires reasoning about another. Often, the service logic and user interface are coded in the same general purpose language, which can provide compile-time checks about the consistent use of module interfaces. However, the interface through which the user interface is specified may be too low-level and dynamic to say anything meaningful about the flow of information between service and user at compile-time (witness the Tk widget set).

Mawl's division of a service into service logic code and user interface code allows a great deal of consistency checking to be performed at compile time. First, the service logic and the MHTML may be independently analyzed to ensure that they are internally consistent. For the service code, this means standard type checking and semantics checking. For MHTML, this means verifying that a template is legal MHTML.

Additionally, the service logic (i.e., Greet.mawl in Figure 3) and the MHTML templates may be checked against one another. The form abstraction makes this possible by providing a type signature expressing the structure of a service/user interaction. The MHTML represents the body of a form's put method and can be analyzed to ensure it is consistent with respect to the form's type signature.

For example, Figure 3(b) shows the content of the file GetName.mhtml, which is the MHTML template associated with GetName form. This template contains no uses of the MVAR mark and contains one INPUT mark named id, which is consistent with the

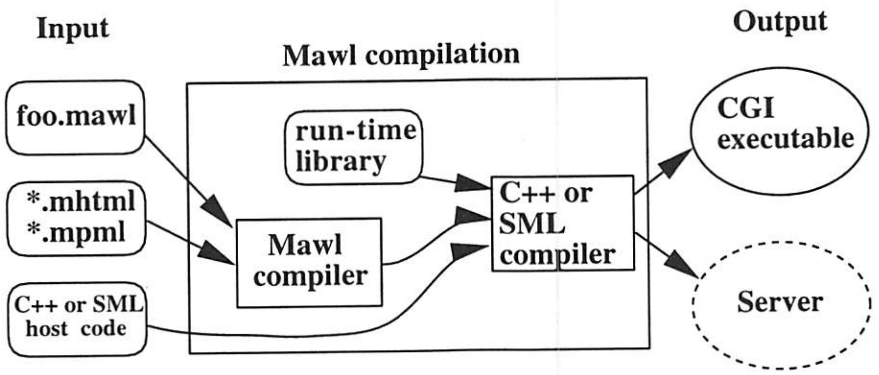


Figure 4: The mawl compilation process.

type signature of the associated `GetName` form. Similarly, the template in Figure 3(c) agrees with the `ShowInfo` form, since the template has `MVAR` marks referring to the service data `id`, `cnt`, and `time` and has no `INPUT` marks.

Another example of a consistency check is to ensure that only values of list type are used in `MITER` marks. It is not required that the MHTML refer to all the service data passed to a form, which is useful for multi-device services, as discussed in Section 3.4.

Our combination of a declarative languge (MHTML) with a sequential imperative language (mawl) can be thought of as a language embedding, such as SQL in C. The form uses a functional interface to moderate between the two languages. The HTML language was extended (to MHTML) so that values in the mawl type system can be substituted into MHTML and so that static checking can be performed on MHTML and between mawl and MHTML. This degree of compile-time checking is much greater than traditionally found in embedded languages. Of course, this comes at a price: construction of a parser and semantics checker for MHTML as well as for mawl. For a more complicated embedded language, this may be too great a luxury to afford.

3.2 Implementation Flexibility

A main advantage of the mawl architecture is implementation flexibility, which is realized by having service logic centrally specified in a session and via the form, which identifies the point at which a session relinquishes control to the user and at which control returns to the service.

Figure 4 shows the mawl compilation process, to which there are three inputs: the logic of the service, written in mawl; document templates, written in MHTML or MPML (more on this markup language in Section 3.4); and support code written in a host language. The mawl compiler takes the first two inputs, which pass through the traditional compiler steps of lexing, parsing, semantic checking, and code generation. The mawl compiler back end generates code in the host language. Then this code is compiled by the host language compiler along with the input support code. Currently supported host languages are C++ [Str86] and Standard ML of New Jersey [MA91]. Support for Java [GA96] is planned. A compiled mawl service is linked with a run-time library to form a complete executable.

A service can be compiled either into a CGI executable or into a stand-alone server. In the CGI implementation, when a session sends out HTML via a form `put`, execution of the session is suspended and the session-instance (`auto`) state is stored on disk or in a database. This is necessary since the CGI process terminates once the HTTP request that started the process has been fulfilled. When a response is received, execution of the session picks up from the point of suspension, with the session-instance state restored. Mawl encodes the session instance in a unique identifier that is stored in the `ACTION` field of the `FORM` mark inserted into the HTML by mawl. Execution of the session continues until another form is encountered, sending another document to the user. Once a session ends, the storage for session-instance state is released. In the server implementation, each session instance is a thread and the compiled `put` method simply suspends after sending the HTML. Again, the identity of the thread is embedded in the HTML so that the server can determine which thread to awaken upon receiving another HTTP request, or if it needs to start a new thread.

Returning to our comparison between the mawl form and interface definition languages, we see that the mawl compiler acts very much like an IDL compiler. An IDL compiler takes an IDL program and produces the code to manage the transfer of data between the sender of a message (client) and the receiver (server). In the mawl programming model, the sender of a message is the web service, although this "send" operation compiles into a code whose function is to respond to an HTTP request. Thus, mawl reverses the roles of client (browser) and server (web service).

As with many IDL compilers, the mawl compiler performs little transformation or optimization, assuming that the transport medium is slow. Mawl has a further advantage because a form is presented to a human to fill out, which lengthens the delay considerably. However, as we will see in Section 3.5, one advantage of the form abstraction is that it allows mawl services to interact with other web services. Thus, recent work on optimization for IDL compilers [EFF$^+$97] might be applicable to mawl for such settings.

3.3 Prototyping Services

Section 3.1 discussed the advantages of compile-time checking of a service against MHTML, which is possible because of the separation of service logic and user interface via the form abstraction. In our experience with mawl at Bell Labs, we have found that programmers sometimes balk at using a statically typed language for web service programming, complaining that type checking impedes rapid prototyping. The requirement that a service and its MHTML type check conflicts with the demands of "Internet time", which requires that services be prototyped and deployed quickly. Programmers often refer to the advantages of type-free languages such as perl, tcl, and ksh, which are traditionally used to program CGI scripts. These languages support prototyping by offering fast turnaround in the compile-edit-debug cycle, as they perform no semantic analysis and are interpreted.

Compounding this problem was the fact that the initial implementation of mawl was overly restrictive, requiring an MHTML template for every form declared in the service logic. To address this, the mawl compiler was modified so that the service logic language could be compiled, executed and tested without any MHTML templates. This required no change to the mawl language. The mawl compiler now generates a default MHTML template when none exists for a form, using the form's type signature (more on this below). Thus, as a soon as a service compiles, a user can interact with it via a web browser.

With the new implementation, we get the best of both worlds. Static type checking not only prevents a large class of run-time errors in mawl services, but also assists in prototyping since the programmer is not required to code MHTML. This points to an interesting measure for a domain-specific abstraction: does the abstraction capture some *necessary* part of the domain? In the domain of form-based services, a service programmer must decide, for each service/user interaction, what information will be exchanged between service and user. This decision is totally independent of the implementation language but is absolutely necessary in order to build a service. In general purpose languages such as perl, tcl, and ksh, the way in which this "signature" is encoded can vary widely and may be quite dynamic. With mawl, the form abstraction (via its static type signature) captures this information precisely in a localized, analyzable construct.

Developer experience with this feature has been quite positive. Mawl's static type system allows the execution of "incomplete" programs that do not contain some or any MHTML. In languages such as perl, the lack of a type system means that incomplete programs are not possible, forcing the programmer to specify some behavior for the incomplete part. The form type signature enables the automatic generation of MHTML, in addition to providing compile-time guarantees when MHTML is present. Using static types to specify "user interface types" can be a boon to prototyping.

3.3.1 Deriving MHTML from Type Signatures

We now discuss some of the issues in generating default MHTML from form types. The essence of a form is that it expresses the flow of information from service to user and back. However, it does not express any coupling between the outgoing and incoming data that is often expressed in user interfaces. For example, given a form type signature

```
{ [int] intlist } -> { int i }
```

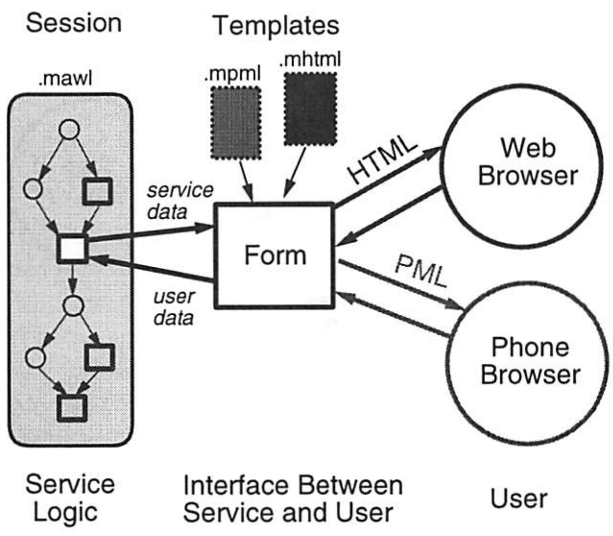


Figure 5: Supporting multiple devices with the same service logic.

what user interface should be generated? There are at least three possible interpretations:

- select an integer from a list of integers, returning the selected integer
- select an integer from a list of integers, returning the index of the selected integer
- present a list of integers, collect an integer from the user, and return it

The first two interpretations are equivalent in the sense that the returned integer is tightly coupled to the input list, but are clearly distinct from the third. Our current transformation of a form type signature to MHTML does not infer a coupling between the service data and user data. The service data is displayed via HTML tables and appropriate marks are included to get data from the user.

This points to a possibility for a third sublanguage in mawl, which would express the constraints between the service data and user data of a form. An example constraint might state a user field is "one-of" a list field in the service record. Such constraints would be optional and could serve two purposes: to generate better default user interfaces; to ensure that MHTML, when provided by the programmer, is consistent with the constraints.

3.4 Multi-Device Services

This section shows how the form abstraction allows a service to interact with multiple browsers. In particular, we focus on supporting both the graphical web browser and the telephone. The issues arising in telephone access to services include many of those that will arise when making services available to a large and diverse collection of devices; indeed, it is hard to imagine two user interfaces more dissimilar than the telephone and the graphical web browser.

The mawl architecture supports multiple devices by allowing multiple templates to be associated with a form, as shown in Figure 5. Mawl uses the `USER_AGENT` of the requester to determine whether to use an HTML template (for the web browser) or a PML template (for the telephone browser). The Phone Markup Language (PML) is a superset of HTML, extended to describe content for interpretation over a telephone. The telephone browser is provided by a system called TelePortal, developed at Bell Labs. TelePortal fetches documents from the web, and "reads" them over the telephone via interactive voice response (IVR) systems. It can also collect data from a user (typically via touchtone or automatic speech recognition).

It is clear that the graphical web browser has a much greater capacity than the telephone browser interface to present and collect data. It is an easy translation to take an IVR service and turn it into a (rather dull) web service. The other direction presents some interesting difficulties, as discussed below. In general, telephone access to a web service will typically offer a limited subset of the functionality available from a graphical web browser.

We have built a number of services that are accessible via both the web and telephone.[5] In all cases, there is one service specification that drives all devices–only the templates change. A self-service banking application (such as the Any-Time Teller) requires essentially the same set of forms for both web and telephone interfaces. However, the presentation of information and collection of the information differs radically, as suited to the device. In general, a given interaction over the telephone will present less information and collect less information than the corresponding interaction over the web.

[5]For a demonstration of a prototype banking service (the Any-Time Teller) with both web and telephone interfaces, visit
`http://www.bell-labs.com/projects/MAWL/anytime.html`

As a concrete example, consider the "login" form of the Any-Time Teller, which has type signature

```
{} -> { string name, int acctid, int pin }
```

On a web browser, an HTML page with three input fields corresponding to the three record fields above is presented: the user may enter either her name or account id, and a PIN to login. Entering alphabetic characters over the telephone touchpad is tedious and error-prone. Thus, the login form should prompt the user only for an account id and PIN, which are integers. This is accomplished by having two different templates. The MPML template uses the "hidden" attribute for the `INPUT` mark for `name`. As a result, TelePortal (the telephone browser) does not prompt the user for a name, but does return a null value for the field. The MHTML template does not use the hidden attribute, so that an input field appears for the `name` attribute.

There are many other ways to support multiple devices when programming services. For example, a general purpose language such as Java, supported by the Java Virtual Machine, allows a multitude of devices to be programmed in a single language. The Inferno operating system [SMD97] represents a similar approach, but starts with the operating system as the common denominator rather than a language. While such work definitely improves the state of programming for heterogeneous collections of devices, the question of a software architecture for service creation is left open. Form-based services will continue to be prominent even as the ends of the network become smarter. We have started to explore mawl services in which a form (or set of forms) is represented by a Java applet, which would allow mawl program to interact with Java-enabled devices.

3.5 Composing Web Services

A simple but powerful attribute of the web is that new web pages can be easily linked to existing web pages. It is similarly desirable to build new web services by combining, collating, or re-presenting information from other web services. An example of such a service is the MetaCrawler [SE95], which collates results from several search engines. Another example is a web service through which customers can order products. This web service might query a courier service (such as FedEx) to present the status of an order to the user.

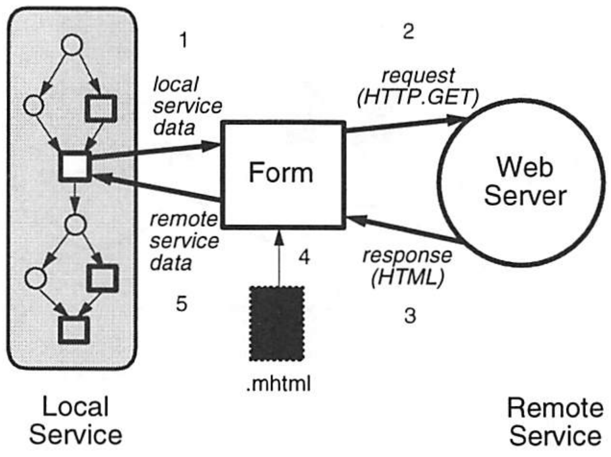


Figure 6: Using the mawl architecture to interact with remote web services. Integer labels show flow of data. Compare the use of the template with Figure 2

It is possible to compose web services like these using existing programming tools such as CGI scripts. However, the programming work is tedious, as it involves the sending of low-level HTTP messages and the parsing of HTML documents to obtain the information of interest.

The form abstraction (along with templates) allow mawl services to interact with other web services, as illustrated in Figure 6. We contrast the data flow and use of templates in this figure with that of Figure 2, where the service is interacting with a web browser. We will use the term "local service" to refer to the service the programmer is creating. As shown in Figure 2, a form interacts with a web browser by combining service data with a template to create HTML that is sent to the web browser (as a result of the current HTTP request), and receives user data in response (in the form of the next HTTP request). However, as shown in Figure 6, a form interacts with a remote web service by sending an HTTP request to the remote service, parameterized with local service data, and extracting remote service data from the HTML document returned by the remote service.

A template is used to extract the remote service data from fields in the HTML document. That is, MHTML is used as a language for pattern matching against an HTML document. When MHTML is used to generate HTML to send to a web browser (Figure 2), the `MVAR` marks specified where to substitute service data into the template. Now, MHTML is used to parse the HTML sent back from the remote web service. In this case, the `MVAR` mark of MHTML is used to bind values in the HTML document to fields in the form's return record. Thus, the `ShowInfo.mhtml` template in Figure 3(c) can be used to extract the name, count and time information from an HTML document of this structure.

The implementation of this feature requires only that a new `query` method for forms be added to mawl. To access the remote service, `query` is used instead of `put`. The method `query` takes as input the URL of a remote service, and a record of local service data. Invoking the method causes an HTTP request to be sent to the remote web server. The received HTML is then matched against with the MHTML template in order to extract the relevant data.

3.6 Usage Analysis

While tools for the construction of web sites and services are numerous, most of these tools lack support for the later parts of the software life cycle: the analysis of a service, and the subsequent modification of the service. By analyzing usage, a service provider can restructure a service to meet the needs of users better, or improve its performance. For example, analysis of a service might show that users routinely follow paths through a service that are more complicated than necessary. By identifying the pattern and restructuring the service, providers can improve their services.

Ideally, the analyses of service/user interactions should come for free as a side-effect of a service. Such logging may be difficult to achieve if services are programmed in an ad-hoc fashion, where the flow of information between service and user is not clear. Mawl's form abstraction provides a centralized point at which to monitor the interactions between service and user. A flag to the compiler or run-time system enables logging of each interaction.

3.6.1 Logging

When a session invokes a form, instrumentation records the service data sent to the form, the template used to create the user interface, other session-specific information (such as the session identifier and current source line), as well as timing information. When a user response to a form arrives, instrumentation records the user data returned to the session. With forms, we can record not only the amount of time between a request and a response, but the amount of time between the response to one form request and the next. This allows measurement of service performance.

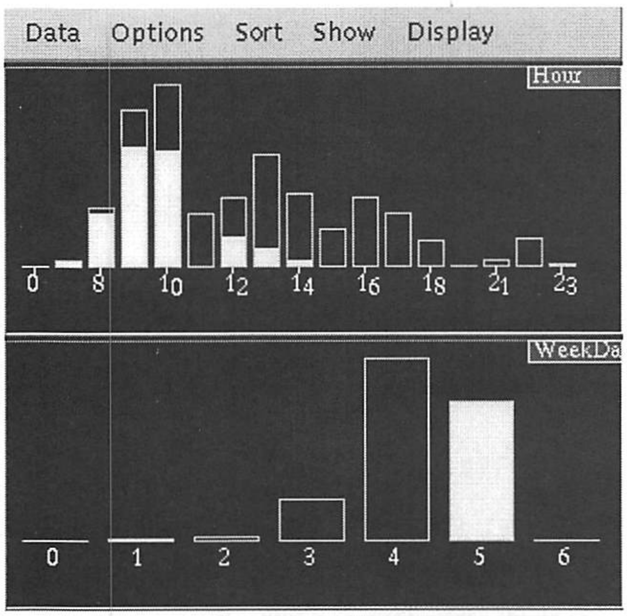


Figure 7: A bar chart view of LunchBot usage.

The template abstraction allows the data passed to and from the form to be related to the user interface that the user views. This is due to the precise mapping between service logic variables and the dynamic portions of the templates (as specified via the `MVAR` marks). This is especially useful for restructuring the templates based on data profiles. For example, profiling the user data returned by a form might show that a particular item in a `SELECT` menu was very popular. Such profile information could be used to reorder the items in a `SELECT` accordingly.

3.6.2 Visualizing the LunchBot

The Mawl system includes a data visualization component called PathView, which is a Java applet that displays user interactions with a service as paths through a graph. PathView enables analysis of the user paths in conjunction with other service statistics, using multiple views to allow exploration of various facets of user behavior. An extensible relational data interface allows new sources of data to be incorporated easily into an analysis. As an example, we use PathView to analyze the usage of the LunchBot, a web service for ordering weekly group lunches. The LunchBot was implemented in mawl shortly after the language was developed.

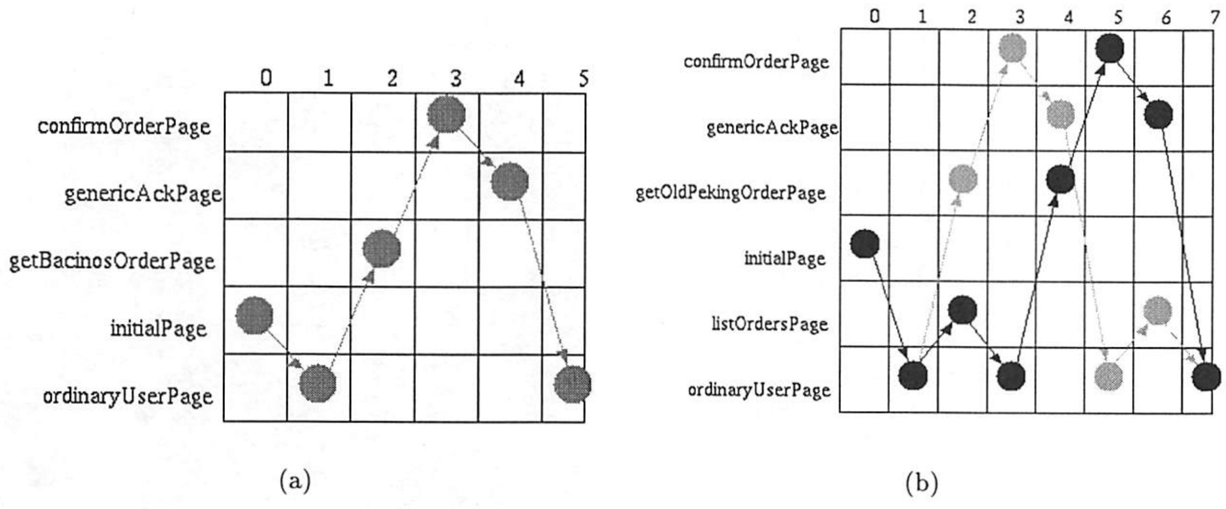


Figure 8: A typical LunchBot scenario: ordering lunch (a). The "list orders" variant (b).

When do users access the LunchBot? The bar chart in Figure 7 shows the amount of activity by hour and weekday. The Lunchbot gets most of its use on Thursday and Fridays (days 4 and 5) every week; this is as expected, since all lunch events in this period were on Fridays. Usage on Friday is highlighted. From the hour bar chart, we can see that the major use of the Lunchbot occurs Friday morning. Most people wait until the last two hours before lunch on Friday to order, after the final warning message is sent (usually around 9 AM) announcing the imminent close of the Lunchbot.

One goal of path analysis is to identify common sequences of user interactions and tune the services to create "shortcuts" for these scenarios. Figure 8(a) shows a common pattern in the LunchBot: the "order lunch" mountain peak. With further analysis, we find a large set of paths that contain both the "order lunch" peak and a "list orders" hill, as shown in Figure 8(b). It turns out that users often list orders before ordering lunch (blue/black path). Other users list orders after ordering lunch (yellow/grey path). The frequent occurrence of the (list orders, order lunch) sequence suggests that users like to see what items are popular (a complicating factor is that the list orders page also lists the user associated with each order; a pessimist might infer that people examine the orders to see if they want to attend lunch at all; we prefer the optimistic analysis). Annotating the favorite items on the menu would provide a simple shortcut replacing the more complicated sequence of interactions.

4 Summary

Mawl was created to address two specific problems in the creation of dynamic web services: the lack of compile-time guarantees about of services, and the low-level of programming involved in coding to the CGI model. A language was necessary in order to provide these guarantees and give implementation flexibility. Neither libraries nor macros provide solutions to these two problems.

The form is the basic abstraction that helped to solve these two problems, by enforcing a separation of concerns between service logic and user interface descriptions. The mawl service architecture and form abstraction have been quite stable since the language's inception and have been used to solve several new problems quite different in nature from the initial two: prototyping and composing services, accommodating multiple devices, and enabling usage analysis. The only solution that required a change to the language (and a minor change, at that) was the composition problem. All the other solutions only changed the compiler analysis or run-time infrastructure.

5 Availability

The mawl language (version 2.0, with C++ as a host language) is available at

`http://www.bell-labs.com/projects/MAWL/`

for SGI and Solaris platforms. Mawl is part of a larger project called Tardis, which includes TelePortal, the platform by which interactive voice response systems may be programmed using HTML. TelePortal is not currently available.

6 Acknowledgments

Christopher Ramming and David Ladd are the originators of the mawl language. Thanks to Natasha Tatarchuk for her work on the visualization applets. Thanks also to Mooly Sagiv and Mike Siff for their perceptive comments and recommendations.

References

[Aea97] D. Atkins and T. Ball et al. Integrated web and telephone service creation. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[BL95] T. Berners-Lee. Hypertext transfer protocol (HTTP/1.0). *Working Group of the Internet Engineering Task Force*, October 1995.

[BLC95] T. Berners-Lee and D. Connolly. Hypertext markup language (HTML 2.0). *Working Group of the Internet Engineering Task Force*, August 1995.

[EFF$^+$97] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing idl compiler. In *Proceedings of the 1997 Conference on Programming Language Design and Implementation (to appear)*, June 1997.

[GA96] J. Gosling and K. Arnold. *The Java Programming Language.* Addison-Wesley, 1996.

[Gro95] Object Management Group. The common object request broker: Architecture and specification. Technical Report Edition 2.0, July 1995.

[Kor86] D.G. Korn. Ksh—a shell programming language. Technical report, AT&T Bell Laboratories, 1986.

[LR95] D. A. Ladd and J. C. Ramming. Programming the web: An application-oriented language for hypermedia service programming. In *Proceedings of the 4th International World Wide Web Conference*, pages 567–586. World Wide Web Consortium, December 1995.

[MA91] D. B. McQueen and A. Appel. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 1–2. Springer-Verlag, 1991.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[SE95] E. Selberg and O. Etzioni. Multi-engine search and comparison using the Metacrawler. In *Proceedings of the 4th International World Wide Web Conference*, pages 195–208. World Wide Web Consortium, December 1995.

[SMD97] R. Pike et al. S. M. Doward. The inferno operating system. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[Sri95] R. Srinivasan. Remote procedure call protocol specification version 2. Technical Report Technical Report RFC 1831, Sun Microsystems, August 1995.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[WS90] L. Wall and R. L. Schwartz. *Programming PERL.* O'Reilly & Associates, 1990.

Experience with a Language for Writing Coherence Protocols

Satish Chandra[1], Michael Dahlin[2], Bradley Richards[3], Randolph Y. Wang[4]
Thomas E. Anderson[4] and James R. Larus[1]

[1] *University of Wisconsin, Madison*
[2] *University of Texas, Austin*
[3] *Vassar College*
[4] *University of California, Berkeley*

Abstract. In this paper we describe our experience with *Teapot* [7], a domain-specific language for writing *cache coherence protocols*. Cache coherence is of concern when parallel and distributed computing systems make local replicas of shared data to improve scalability and performance. In both distributed shared memory systems and distributed file systems, a *coherence protocol* maintains agreement among the replicated copies as the underlying data are modified by programs running on the system.

Cache coherence protocols are notoriously difficult to implement, debug, and maintain. Unfortunately, protocols are not off-the-shelf items, as their details depend on the requirements of the system under consideration. This paper presents case studies detailing the successes and shortcomings of using Teapot for writing coherence protocols in two systems. The first system, *loosely coherent memory* (LCM) [16], implements a particular type of distributed shared memory suitable for data-parallel programming. The second system, the *xFS distributed file system* [9], implements a high-performance, serverless file system.

Our overall experience with Teapot has been very positive. In particular, Teapot's language features resulted in considerable simplifications in the protocol source code for both systems. Furthermore, Teapot's close coupling between implementation and formal verification helped to achieve much higher confidence in our protocol implementations than previously possible and reduced the time to build the protocols. By using Teapot to solve real problems in complex systems, we also discovered several shortcomings of the Teapot design. Most noticeably, we found Teapot lacking in support for multithreaded environments, for expressing actions that transcend several cache blocks, and for handling blocking system calls. We believe that domain-specific languages are valuable tools for writing cache coherence protocols.

1 Introduction

Cache coherence engines are key components in several parallel and distributed computing systems. Coherence is of concern whenever distributed systems make local replicas of shared information for reasons of performance or availability (or both), because systems must keep replicas current as they modify the shared information. Thus, distributed shared memory systems [6,15], distributed file systems [9,20], and high-performance client-server database systems [12] all implement cache coherence protocols. Coherence in web caching is also a current research topic in the distributed systems community [19].

Tools that facilitate the implementation of cache coherence protocols are important for two reasons. First, coherence protocols, while ubiquitous, show a great deal of variety because the protocol for a particular system is closely linked to its sharing semantics and performance goals. For example, different distributed shared memory systems provide different memory consistency models [13], which support different assumptions that application programs can make about the currency of cached values. Moreover, systems with similar sharing semantics can have vastly different protocols that use different algorithms to achieve the same task, albeit with different performance considerations. Thus, each system essentially needs its own coherence protocol.

Second, and perhaps more importantly, cache coherence protocols represent complex, distributed algorithms that are difficult to reason about, and often contain subtle race conditions that are difficult to debug via system testing. Furthermore, to our knowledge, previous systems have not attempted a clear separation between the cache-coherence engine and other implementation details of the system, such as fault management, low-level I/O, threads, synchronization, and network communication. It is not difficult to imagine the hazards of this approach. The implementor cannot reason about the coherence protocol in isolation from other details, and any modification she makes in the system can potentially impact the protocol's correctness—a debugging nightmare. Experimentation with newer protocols is a perilous proposition at best.

Teapot is a protocol writing environment that offers two significant improvements over writing ad-hoc C code. First, it is a domain-specific language specifically targeted at writing coherence protocols. As such, it forces a protocol programmer to think about the logical structure of a protocol, independent of the other entanglements of a system. The language features of Teapot easily express the control structures commonly found in coherence protocols. Second, Teapot facilitates automatic verification of protocols because it not only translates Teapot protocols into executable C code, but also generates input code for MurΦ, an automatic verification system from Stanford [10]. MurΦ can then be used to detect violations of invariants with a modest amount of verification time. For example, our system might report a stylized trace of a sequence of events that would cause a deadlock. A protocol can be run through a verification system prior to actual execution, to detect possible error cases, *without* having to manually rewrite the protocol in MurΦ's input language.

The Teapot work was originally undertaken to aid protocol programmers for the Blizzard distributed shared memory system [25]. Blizzard exports a cache-coherence protocol programming interface to an application writer, so she can supply a coherence protocol that best suits the requirements of her application. Writing such protocols in C, without domain-specific tools, turned out to be a difficult task, fraught with problems of deadlocks, livelocks, core dumps, and most annoyingly, wrong answers. After a few initial protocols (all variants of conventional shared memory protocols) were successfully developed using Teapot, the Blizzard team at Wisconsin wrote several other, more complicated coherence protocols for their system. We report on one such protocol here. Subsequently, the xFS team at UC Berkeley adopted Teapot to write the coherence protocol for their distributed file system. As expected, these teams encountered several rough spots, because the original Teapot design did not anticipate all of the requirements of other protocols in the context of Blizzard, much less those arising in a distributed file system context.

This paper describes our experiences with using Teapot to implement the coherence engines in two distinct systems. In both systems, we found Teapot to be vastly superior to earlier efforts to implement the protocols using C without domain-specific tools. The paper makes several contributions. First, it highlights the aspects of Teapot that proved successful across several protocols:

- *Domain-specific language constructs*, such as a state-centric control structure and continuations, simplified the protocol writing task.
- *Automatic protocol verification* using the MurΦ system improved system confidence and reduced testing time.

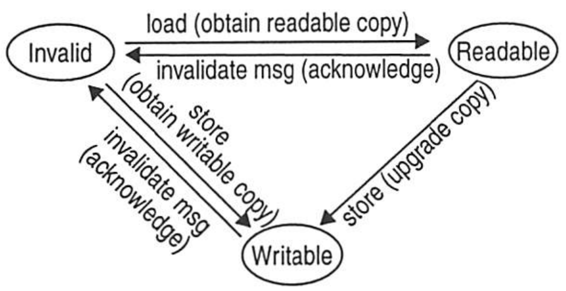
Perhaps more importantly, this paper also discusses shortcomings of the language that became apparent only when we attempted to develop protocols that were more complicated than the simple protocol examples on which Teapot was originally tested. In particular, our experience indicates that improved support for multi-threaded environments, for protocol actions that affect multiple blocks, for local protocol actions that might block, and for automated verification test strategies would further ease the job of a protocol designer. Finally, the paper generalizes our experience to provide guidelines for future domain-specific languages for systems software.

The rest of the paper is organized as follows. Section 2 provides some basic background on cache coherence protocols and describes the implementation problems generally faced by protocol programmers. Section 3 introduces the language features in Teapot that address the difficulties presented in Section 2. Section 4 presents the case-study of LCM, and Section 5 presents the case-study of xFS. Section 6 describes some related work. Section 7 concludes the paper with implications for domain-specific languages for systems software.

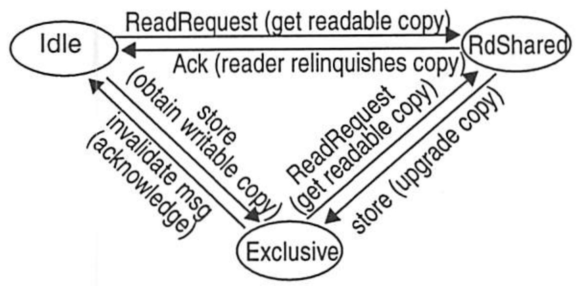
2 Coherence Protocols and Complications

In systems with caching, read operations on shared data typically cache the value after fetching it from remote nodes, in the expectation that future read references will "hit" locally. Write operations on shared data must take steps—coherence actions—so readers with cached values do not continue to see the old value indefinitely. This section describes coherence protocols in more detail in the context of distributed shared-memory systems, though the issues discussed apply equally well to other contexts with appropriate changes in terminology.

Shared-memory systems can be implemented using a pair of mechanisms: access control and communication. Access control allows the system to declare which types of accesses to particular regions of memory are permitted. These permissions typically include: no access (*invalid*), reads only (*readonly*), and both reads and


Invalid
Readable
Writable
load (obtain readable copy)
invalidate msg (acknowledge)
store (obtain writable copy)
invalidate msg (acknowledge)
store (upgrade copy)

(a)


Idle
RdShared
Exclusive
ReadRequest (get readable copy)
Ack (reader relinquishes copy)
store (obtain writable copy)
invalidate msg (acknowledge)
ReadRequest (get readable copy)
store (upgrade copy)

(b)

Figure 1: Idealized protocol state machine for (a) the non-home side, and (b) the home side. Transitions are labeled with causes and, in parentheses, actions.

writes (*readwrite*). Performing an illegal access (for example, writing a *readonly* region) causes an *access fault* and invokes the coherence protocol. Communication allows a system to exchange control information and data between processors. The coherence protocol comes into play at an access fault. It must obtain a copy of the referenced data with appropriate access permissions and satisfy the access. Many protocols designate a *home node* that coordinates accesses to a particular range of memory addresses. The faulting processor sends a request to the home node for a copy of the required data, which responds with the data after updating its bookkeeping information. After receiving the response, the faulting processor typically caches the data so subsequent accesses will succeed without communication.

A common technique for ensuring coherence allows at most a single writer or multiple readers for any block of memory at a time. When the home receives a request for a writable copy of the block, it asks processors currently holding a readable copy to invalidate it, i.e. allow no further accesses. A writable copy can then be sent to the requestor. A cache coherence protocol specifies the actions taken by the home and caching processors in response to access faults and incoming messages. These actions are commonly captured by finite state machines, with transitions between protocol *states* occurring in response to faults and messages. Figure 1 shows sample state machines describing protocol actions for a caching processor and the corresponding home side. Both the home and caching processors associate a state with each memory block. At an access fault or upon a message arrival, the protocol engine consults the appropriate block's state to determine the correct action. Typical protocol actions involve sending messages and updating the state, access permissions, and contents of a memory block. Home nodes also maintain a *directory*, a per-block data structure that usually keeps track of which processors have a readable copy, or which processor has an exclusive copy.

As an example, consider a (non-home) block that is initially in the *Invalid* state. A processor reading any address within the block causes an access fault, at which time the protocol is invoked. Its action is to send a request to the home node for a readable copy and await a response. Assuming no outstanding writable copy exists (the *Idle* state in Figure 1), the home responds with a readable copy and changes its state to *ReadShared.* The arrival of this message on the non-home side causes the protocol to copy the incoming data to memory and change the block's state to *Readable* (and access permissions are changed from *invalid* to *readonly*).

Unfortunately, specifying protocols is much more difficult than the simple three-state diagrams in Figure 1 would lead one to believe. The main difficulty is that, although the transitions shown *appear* to be atomic, many state changes in response to protocol events cannot be performed atomically. Consider the transition from the *Exclusive* state to the *ReadShared* state in Figure 1. Conceptually, when a request arrives in the *Exclusive* state for a readable copy of a block, the protocol must retrieve the exclusive copy from the previous owner and pass it along to the requestor. The protocol sends an invalidation request to the current block holder, and must await a response before proceeding. But, to avoid deadlock, protocol actions must run to completion and terminate. This requires that an intermediate state, *Excl-To-ReadShared* (*Excl-RS* for short), be introduced. After sending the invalidation request, the protocol moves to the *Excl-RS* state and relinquishes the processor. When the invalidation acknowledgment arrives in this intermediate state, the processor sends a response to the original requestor and completes the transition to *ReadShared.* A revised state diagram incorporating the required intermediate states is shown in Figure 2 (which is still far removed from a realistic protocol).

Introducing intermediate states increases the number of states a programmer has to think about. Furthermore, while in an intermediate state, messages other than the expected reply can arrive. For example, before the inval-

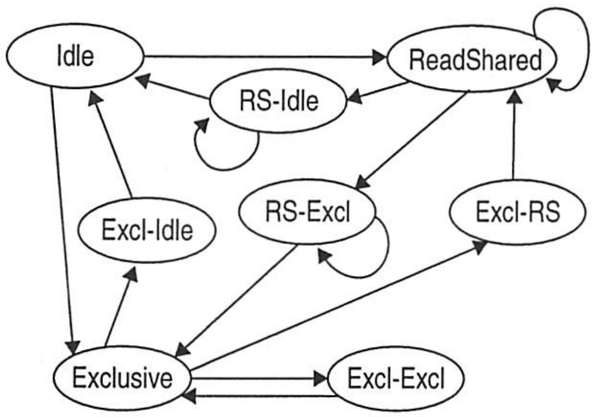


Figure 2: State machine (home side) with intermediate states necessary to avoid synchronous communication.

idation response arrives in the *Excl-RS* state, another request for an exclusive copy could arrive from a different processor. A protocol designer must anticipate the arrival of such unsolicited messages and handle them in an appropriate manner. It may be tempting to not take such messages out of the network while they are not welcome: this, however, is not an option on most systems, because messages must constantly be drained from the network to avoid deadlock in the network fabric [27].

Message reordering in the network adds to the woes of a protocol programmer. For example, processors may appear to request copies of cache blocks which they already have, if a read request message overtakes an invalidation acknowledgment message in the network. The protocol might have to await delayed messages before deciphering the situation and determining the correct action. Without machine assistance, anticipating all possible network reorderings is a very difficult task!

The traditional method of programming coherence state machines usually resorts to ad-hoc techniques: unexpected messages may be queued, they may be negatively acknowledged (nack'ed), or their presence may be marked by a "flag" variable. Additional flag variables are often used to track the out-of-order arrival of messages as well. These techniques invite protocol bugs. Queuing can easily lead to deadlocks; similarly, nack'ing can lead to livelocks or deadlocks. Flag variables are essentially extra protocol state—failing to update or test a flag at all the right places again leads to correctness problems. Moreover, protocols implemented in this style are very difficult to understand and modify.

The case studies presented in sections 4 and 5 show that all these complications were serious issues in the initial state machine versions of those protocols. In the next section, we highlight the features of Teapot that aid a protocol programmer.

3 Teapot

The Teapot language resembles Pascal with extensions for protocol programming support, but fewer built-in types. Space does not permit a complete description of the language; the reader is referred to the original paper [7] for further language details. The Teapot compiler can generate executable C code from a protocol specification, and can also translate it to code that can be fed to the MurΦ verification system [10].

```
State Stache.Home_Exclusive{}
Begin
    Message GET_RO_REQ(id:ID; Var info:INFO; src: NODE)
    Var
        itor : SHARER_LIST_ITOR;
        j : NODE;
    Begin
        Send(GetOwner(info), PUT_DATA_REQ, id);
        IncSharer(info, src);
        Suspend(L, SetState(info, Home_Excl_To_Sh{L}));
        -- send out a readable copy to all nodes that want a copy
        -- (more nodes might want a copy while you were waiting)
        Init(itor, info, NumSharers(info));
        While (Next(itor, j)) Do
            SendData(j, GET_RO_RESP, id, TPPI_Blk_No_Tag_Change);
        End;
    End;
    -- other messages ...
    Message DEFAULT(id:ID; Var info: INFO; src: NODE)
    Begin
        Error("Invalid message %s to Home_Exclusive",Msg_To_Str(MessageTag));
    End;
End;
```

Figure 3: Teapot example

```
1.  State Stache.Home_Excl_To_Sh{C:CONT}
2.  Begin
3.      Message PUT_DATA_RESP (id: ID; Var info: INFO; src: NODE)
4.      Begin
5.          RecvData(id, TPPI_Blk_Validate_RW, TPPI_Blk_Downgrade_RO);
6.          SetState(info, Home_RS{});
7.          Resume(C);
8.      End;
9.      -- other messages
10.     Message DEFAULT (id: ID; Var info: INFO; src: NODE)
11.     Begin
12.         Enqueue(MessageTag, id, info, src);
13.     End;
14. End;
```
Figure 4: Teapot example (cont'd)

3.1 Language Features

A Teapot program consists of a set of states; each state specifies a set of message types and the actions to be taken on receipt of each message, should it arrive for a cache block in that state. We exhibit some of the features of Teapot using an example. The Teapot code in Figure 3 implements coherence actions for a block in the *Exclusive* state at the home node. Suppose the block receives the request message GET_RO_REQ, asking for a readable copy. The action code for this message first sends a PUT_DATA_REQ message to the current owner (note that the variable info is a pointer to the directory data structure). Next, it executes a Suspend statement. A Suspend statement is much like a "call-with-current-continuation" of functional programming languages. Syntactically, it takes a program label (L), and an intermediate state (Home_Excl_To_Sh) which it visits "in transition". The second label, {L}, specifies where execution should resume upon return, and can differ from the first argument. Operationally, Suspend saves the environment at the point it appears in a handler body and effectively puts the handler to sleep. This mechanism is used to provide a blocking primitive inside a handler, which physically needs to relinquish the processor every time it is invoked

What happens in the intermediate state? Figure 4 shows the Teapot code executed when a PUT_DATA_RESP message arrives. The handler receives the up-to-date content of the cache block from the network, sets its own state to *ReadShared*, and executes a Resume statement. The Resume is the equivalent of a "throw" for a "call-with-current-continuation" of functional programming. Syntactically, it takes a continuation parameter (C) as an argument. (Note from line 1 in Figure 4 that the continuation variable C is a state parameter and is a part of the environment visible to all the message handlers in that state.) Operationally, it restarts a suspended handler immediately after the Suspend statement whose label is captured in C. Thus, after the Resume statement, GET_RO_RESP messages are sent to the set of requesters (see Figure 3 again, lines 13-16). Continuations in Teapot let us avoid having to manually decompose a handler into atomically executable pieces and sequencing them. Further advantages of the Suspend/ Resume primitives are brought out in the case studies.

Teapot provides a mechanism for handling unexpected messages by queuing. It does not solve the problem of deadlocks directly, but facilitates deadlock detection via verification. In lines 10-13 of Figure 4, all messages not directly handled (DEFAULT) are queued for later execution—these messages are appropriately dispatched once the system moves out of an intermediate (*transient*) state.[1] Teapot relies on a small amount of system-specific dispatch code to deliver incoming network messages and previously queued messages, based on a state lookup and the message tag. Note that the DEFAULT messages in Figure 3 flag an error because these messages cannot occur in a correctly functioning system.

3.2 Verification Support

Teapot makes no attempt to verify protocols, but translates protocols into code for the MurΦ automatic verification system [10]. MurΦ explores all possible protocol actions by effectively simulating streams of shared-memory references, and ensuring that no system-wide invariants are violated. If unanticipated messages arrive or deadlock occurs, Teapot transforms the MurΦ error log into a stylized diagram of the protocol events leading to the violation.

Three basic components are required for verification: A MurΦ description of the protocol under test, MurΦ code implementing all types and subroutines used by the protocol, and a *ruleset* describing legal sequences of protocol events. While only the first component is generated by Teapot, examples of the remaining pieces are included with Teapot and can often be reused without modification. User intervention is required only if new

1. Users must declare which states are transient.

types or routines are added, or the protocol being developed only handles stylized streams of protocol events. The latter scenario is described in more detail in the following section.

4 LCM

The Loosely Coherent Memory (LCM) coherence protocol [16] provides sequentially-consistent distributed shared memory as a default, and is similar in many respects to protocols like DASH [18], Alewife [1], and Stache [24]. The key difference is that LCM allows global memory to become temporarily inconsistent under program control. During such phases, a given data item may intentionally have different values on different processors. This makes management of shared data more difficult. Memory is returned to a globally-consistent state by merging distinct versions of each data item and ensuring that all processors see the new values. This requires coordination among all processors in the system, and mixes computation (merge functions) with traditional protocol actions.

LCM implements the semantics of the data-parallel programming language C** [17] faster than conservative, compiler-implemented approaches. C** semantics specify that parallel function invocations on aggregate data do not interact. LCM enforces these semantics by keeping shared-data modifications private until all parallel invocations complete, then returns the system to a consistent state. Processes can still collaborate to produce values via a rich set of reduction operations (including user-specified reductions), but the results of these reductions are not available until after all parallel function invocations finish.

4.1 Initial Implementation

Our first LCM implementation effort was undertaken without the support of any formal methods or tools. The C-code source of the Stache (ordinary shared memory) protocol was available to us, so we used it as a starting point, adding extra LCM functionality as required. In retrospect, starting with Stache was an unfortunate decision. Stache, while a relatively simple protocol design, is still a large and complex piece of software. Adding LCM functionality required both that the behavior of existing protocol states be altered and that new states be added—a difficult proposition for the unaided programmer. Small changes in existing states (and the addition of a new states) often had far-reaching effects that were difficult to fully anticipate.

It took several months for a single graduate student, working full-time, to complete the basic protocol modifications, after which a debugging phase began. It took roughly as long to debug the modified protocol as it did to write it in the first place, since the protocol was riddled with subtle timing-related bugs, the result of the unpredictable effects of our modifications. A suite of applications was used to debug the protocol—each application exercising a new set of path-specific bugs in LCM which had to be isolated, understood, and repaired. It often took days to identify infrequently-occurring bugs, and the resulting "fixes" many times introduced new bugs. Even after the LCM protocol had achieved relative stability, user confidence in its correctness was low.

4.2 Teapot and LCM

An early version of the Teapot system was ready for testing as debugging of the hand-written LCM protocol was being completed, and LCM was reimplemented with Teapot to more thoroughly evaluate the system. The Teapot environment was a vast improvement over the hand-coded approach. We found two language features of Teapot particularly useful: the "state-centric" programming model, and the use of continuations to allow blocking operations in handler code.

Teapot enforces a protocol programming style that is easier to read and debug than that we used in C. Teapot code is organized by protocol states, each of which contains a list of handlers to be run for messages arriving in that state. This contrasts with the handwritten protocol's "message-centric" approach, where large handlers were written for each message type and selected different action code to run based on the protocol's state. Organizing the protocol by states makes it easier to express and implement for several reasons. First, each handler is now a smaller unit of code, since a self-contained handler is written for each combination of message and block state. Second, grouping handlers by state instead of message type keeps related information close together: A state's behavior can be understood by scanning a set of consecutive handlers, instead of looking through the entire protocol. Of course, in retrospect, we could have adopted a state-centric organization in the handwritten protocol, but the C language did not make the benefits of doing so immediately obvious while the Teapot system enforced a disciplined programming style that utilized the better design choice.

Teapot's continuations also made an enormous improvement in handler legibility. Even for handlers using a single `Suspend` statement, keeping the code on either side of the call in the same handler dramatically increased

```
State LCM.Home_Excl {}
   ... other messages
   Message GET_RO_REQ (id: ID; Var info: INFO; src: NODE)
   Begin
      [...]
      If (SameNode(src, GetOwner(info))) Then
         Suspend(L, SetState(info, Home_Excl_To_Idle{L}));
         If (SameState(GetState(info), Home_Idle{})) Then
            SetState(info, Home_RS{});
            AccChg(id, TPPI_Blk_Downgrade_RO);
         Else
            If (InSharers(info, src)) Then
               Suspend(L2, SetState(info, Home_Await_PUT_ACCUM{L2}));
            Endif;
         Endif;
         [...]
      Else
         Send(GetOwner(info), PUT_DATA_REQ, id);
         Suspend(L1, SetState(info, Home_Excl_To_Sh{L1}));
         IncSharer(info, src);
      [...]
      Endif;
   [...]
   End;
```
Figure 5: Teapot handler code containing multiple `Suspend` statements

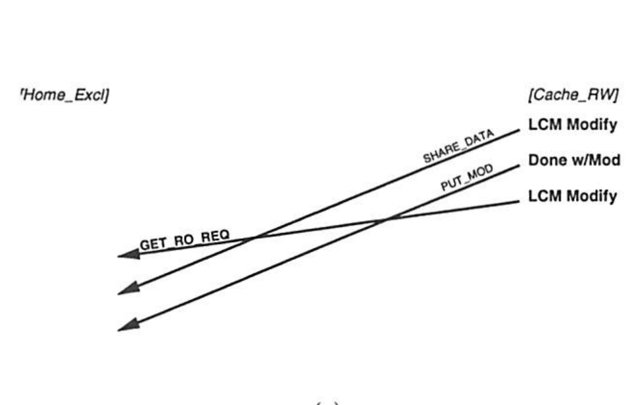
readability. Some handlers used as many as three `Suspend` statements, and therefore had to be split into multiple code fragments in the handwritten version. Figure 5 shows part of an LCM handler with three `Suspend` statements. Without continuations, this code would have been split into at least four distinct handlers making it much harder to write and debug. Teapot also allows dynamic nesting of continuations, a feature used numerous times during the specification of LCM. For example, the first `Suspend` in Figure 5 moves to the `Home_Excl_To_Idle` state, where other handlers (not shown) may suspend again to await delayed messages.

Even with the cleaner design, we uncovered a total of 25 errors using automatic verification. (Each error was fixed as soon as it was detected and understood, and the verification step was repeated.) Many of these were subtle bugs that were unlikely to occur often in practice, but were all the more dangerous as a result. Figure 6 illustrates an LCM bug that is representative of those found through verification. Both diagrams show messages being exchanged between a pair of processors, with time increasing from top to bottom. In each case, a preceding exchange of messages (not shown) has left the cache (non-home) side with the exclusive copy of a given coherence block.

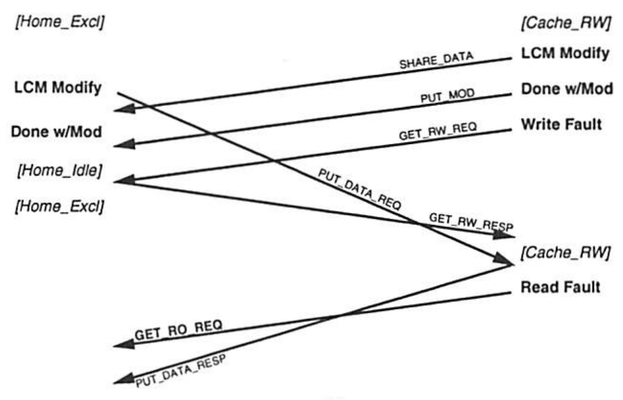
In Figure 6a, the caching processor performs an LCM modification of the block, creating a version that is inconsistent with respect to other copies in the system. However, since the cache side held the exclusive copy at the time it performed the modification, it first sends a *copy* of the block home. This data can be used by the home to respond to requests for the block from other processors. The block is returned home via a `PUT_MOD` message when the cache side is finished. The second LCM modification then faults and requests the block back from the home.[1] Messages have been reordered in the network such that the first to appear at the home is the request for data. The home detects the reordering, since it knows the requestor already *has* a copy of the block. The correct action in this case is to await the `SHARE_DATA` message, then satisfy the request. The home leaves the block in the `Home_LCM` state to denote the fact that at least one processor has created its own version of the block.

Initially, we thought the arrival of the `GET_RO_REQ` in the `Home_Excl` state always implied the message reordering scenario in Figure 6a, and both the hand-written version of LCM and the first Teapot version encoded this assumption. Unfortunately, in the more complicated case shown in Figure 6b, this caused the protocol to respond incorrectly. The home should instead await the `PUT_DATA_RESP` message, transition to the `Home_Idle` state, and satisfy the request. Correcting the protocol is straightforward once the two scenarios have been identified, but it is unreasonable to expect an unaided programmer to have foreseen such a bug, due to the complexity of the cases involved. Enumerating all chains of protocol events and ensuring that they are

1. This scenario arises frequently in applications where a given processor handles several of a set of parallel tasks consecutively.



(a)



(b)

Figure 6: Two different scenarios in which a `GET_RO_REQ` arrives in state `Home_Exclusive`. The appropriate response to the message is different in each case.

properly handled is a job much better handled through verification.

Using Teapot, the new version of the LCM protocol was written, verified, and running applications in two weeks' time. Only one bug was uncovered during field testing of the new protocol, and it occurred in a simple support routine that was intentionally *not* simulated.[1] Also, because of Teapot, we were able to implement easily three variants of LCM: one that eagerly sends updates to consumers at the end of an LCM phase, another that manages extra, distributed copies of some data as a performance optimization, and a version that incorporates both of these features.

4.3 Teapot Shortcomings

While Teapot made it significantly easier to get LCM written and working, it fell short of our needs in several respects. One significant obstacle is Teapot's inability to perform actions across a *set* of blocks. A message handler, for example, can only update the state of the block to which a message is directed. In LCM, action must periodically be taken across a collection of blocks. For example, during a merge phase, a processor returns *all* modified blocks to their homes, where they are combined with copies from other processors. An event handler was written to carry out this flushing operation for a single block, but the handler must somehow be invoked for each block returned. As an application runs, the LCM protocol constructs a list of modified blocks that require flushing at the next reconciliation. This list is traversed when the reconciliation phase begins, and the appropriate event handler invoked on each block. Additional C code was written to traverse the list and invoke handlers in the executable version of the protocol, but this code is outside the scope of the Teapot protocol specification and therefore cannot be verified. The workaround in Teapot was to structure the MurΦ ruleset so that, during a reconciliation, it invoked the handlers for each block in the list. This restructuring significantly increased the complexity of the ruleset and therefore the chances that it could contain an error.

1. The routine was deemed too simple to be hiding any bugs.

Even without operations on sets of blocks, the ruleset for LCM was already much more complicated than those for our previous protocols. Unlike Stache, where any arbitrary stream of interleaved loads and stores to shared memory must be handled, LCM only properly handles stylized sequences of loads and stores. There are distinct phases that all processors must agree to initiate, in which only certain access patterns are legal. Encoding this into a ruleset was a lengthy, complicated, and potentially error-prone process, and represented a significant fraction of the work required to implement LCM. It would be preferable to generate such rulesets automatically from a high-level description of a protocol's memory model, but we currently are unaware of any techniques for doing so.

The last shortcoming was relatively minor. Teapot currently does not allow the testing of a pair of expressions for equality. There were several places in the protocol where pairs of states or node identifiers needed to be compared, and an external routine had to be written to perform these tests. Future releases of Teapot should consider extending the language such that simple comparisons can be done without resorting to external procedures.

5 xFS

xFS, a network file system described in several previous papers [2,9], is designed to eliminate all centralized bot-

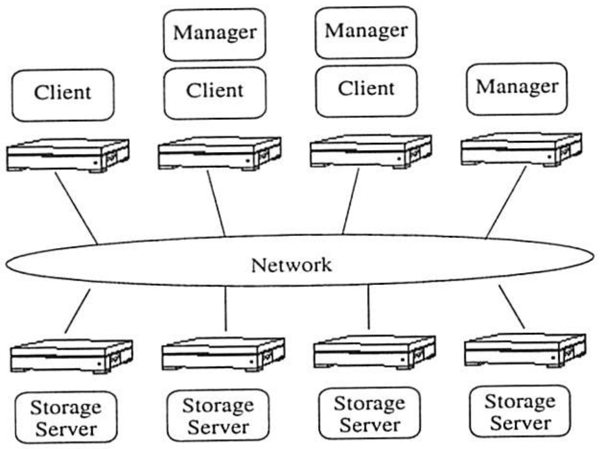


Figure 7: A sample xFS configuration. Clients, managers, and storage servers provide a global memory cache, a distributed metadata manager, and a striped network disk respectively.

tlenecks and efficiently use all resources in a network of workstations. One of the most important features of xFS is its separation of data storage from data management. This separation, while offering superior performance and scalability compared to traditional file systems, also requires a more sophisticated cache coherence protocol. In addition, other aspects of the cluster file system environment—such as multi-level storage and reliability constraints—further complicate the system compared to more traditional DSM coherence protocols. Due to these aspects of the design, we found it difficult to implement a correct protocol with traditional methods. The use of Teapot has resulted in clearer abstraction levels, increased system confidence, and reduced complexity in the implementation of cache coherence in xFS. At the same time, there are significant differences between xFS and the original applications which Teapot was designed to support. These differences have revealed some shortcomings of Teapot.

5.1 Caching in xFS

The three main components of an xFS system are the *clients*, the *managers*, and the *storage servers*. Under the xFS architecture, any machine can be responsible for caching, managing, or storing any piece of data or metadata by instantiating one or more of these subsystems. Figure 7 shows a sample xFS installation.

Each of the three subsystems implements a specific interface. A client accepts file system requests from users, sends data to storage servers on writes, forwards reads to managers on cache misses, and receives replies from storage servers or other clients. It also answers cooperative cache forwarding requests from the manager by sending data to other clients. The job of the metadata manager is tracking locations of file data blocks and forwarding requests from clients to the appropriate destinations. Its functionality is similar to the directory manager in traditional DSM systems. Finally, the storage servers collectively provide the illusion of a striped network disk.

xFS employs a directory-based invalidate cache coherence protocol. This protocol, while similar to those seen in traditional DSM systems, exhibits four important differences that prevent xFS from using previously developed protocols and that complicate the design of xFS. (1) xFS separates data management from data storage. Although this separation allows better locality and more flexible configuration, it splits atomic operations into different phases that are more prone to races and deadlocks. (2) xFS manages more storage levels than traditional DSM systems. For example, it must maintain the coherence of the kernel caches, write-ahead logs, and secondary storage. (3) xFS must maintain reliable data storage in the face of node failures, requiring protocol modifications that do not apply to DSM systems. For example, a client must write its dirty data to storage servers before it can forward it to another client. (4) The xFS client is heavily multi-threaded and it includes potentially blocking calls into the operating system, introducing more chances for synchronization errors not seen in DSM systems.

5.2 Implementation Challenges

The xFS design and environment make the implementation and testing of cache coherence in xFS more difficult than in most systems. The usual problems of proliferation of intermediate states and subtle race conditions were even worse for xFS, as described below.

5.2.1 Unexpected Messages and Network Reordering

An xFS node can receive messages that cannot be processed in its current state. This is also a problem in most DSM coherence systems, but it is particularly pervasive in xFS because xFS separates data storage and control, thereby making it difficult to serialize data transfer messages and control messages with one another: data transfer messages pass between clients and storage servers or between clients and clients, while control messages pass between clients and managers or storage servers and managers.

The xFS protocol also suffers from the message reordering problems mentioned in Section 2. Further compounding the problem, this protocol often allows multiple outstanding messages in the network to maxi-

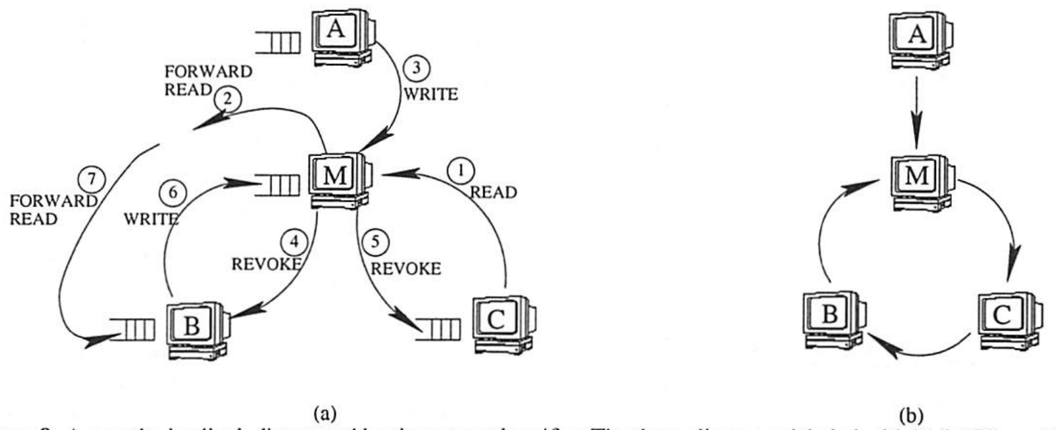


Figure 8: A sample deadlock discovered by the protocol verifier. The three clients are labeled with "A", "B", and "C". The manager is labeled with "M". In Figure (a), arrows denote the directions of the messages. The numbers denote the logical times at which messages are sent and/or received. Shown to the left of each host is a message queue, which holds the requests that are waiting to be processed. Messages that are not queued are processed immediately. In Figure (b), arrows denote the wait-for relationship, and the presence of a cycle indicates a deadlock.

mize performance. For example, an xFS manager does not wait until a client completes a forwarding request to continue, so a subsequent invalidate message can potentially reach the same client out of order. Although such ordering can be enforced at the communication layer [5], recent research has argued that this ordering is best expressed with application state [8]. Furthermore, even if the network ensured in-order messages between nodes, the causes mentioned in the previous paragraph would still require xFS to explicitly handle unexpected message arrivals.

5.2.2 Software Development Complexity

Managing the large number of states needed to implement the xFS state machine was a challenge. Although, intuitively, each block can be in one of only four states—*Read Shared*, *Private Clean*, *Private Dirty*, or *Invalid*—the system must, in fact, use various transient states to mark progress during communication with the operating system and the network. Dealing with unexpected or out of order messages, handling the separation between data storage and data management, maintaining multiple levels of storage hierarchy, and ordering events to ensure reliable data storage increases the number of transient states needed to handle xFS events. Even a simplified view of the xFS coherence engine contains twenty-two states. One needs a systematic approach when dealing with such a large state space.

As we were implementing the protocol, it became clear that the C language was too general. Despite our best intentions, aspects of implementations that were not related to protocol specification were mixed in. The result was less modular, harder to debug, and harder to maintain. Although the xFS protocol is similar to many other DSM protocols, we have found it non-trivial to reuse or modify existing codes, due to their ties to their native environments.

5.3 Teapot and xFS

After several unsuccessful attempts at completing the cache coherence protocol using traditional development methods, we decided to rewrite the system with Teapot. Our experience with this domain specific language has been positive. In particular, the close ties between Teapot and the MurΦ verification system have provided us with an effective testing tool for attacking the problem of unexpected event ordering; many of the bugs we found and corrected would have been extremely difficult to isolate through field testing alone. Furthermore, several aspects of the Teapot language have simplified the engineering complexity in our system.

5.3.1 Testing for Unexpected Event Orderings

Figure 8 shows an example of a bug in an early version of the xFS protocol that would have been difficult to isolate via field testing, but which MurΦ easily discovered. In this version of the protocol, we saw no need for the manager to maintain sequence numbers for its outgoing messages. If a receiver of a manager request was not ready to act upon it, it simply queued it for later processing. MurΦ found the following deadlock bug:

Initially, client B is the sole cacher of a clean block. (1) Client C sends a read request to the manager. (2) The manager forwards the request to client B. To indicate that Client B should send the data to Client C via cooperative caching, the manager also updates its state to indicate that both client B and C are caching the data.

(3) Meanwhile, client A sends a write request to the manager. (4) The manager sends a revoke request to client B, which arrives at client B before the previous forwarding message, invalidating its data. (5) The manager sends a second revoke request to client C, which client C queues, because its requested data has not arrived. (6) Client B sends a write request to the manager, which the manager queues, because its previously sent revoke message has not been acknowledged. (7) The delayed forward message from step 2 finally arrives, which client B queues, because its request to the manager has not been satisfied. Now we have finally reached a deadlock: client A is waiting for the manager to complete the revoke operations; the manager is waiting for client C to acknowledge the revoke request; client C is waiting for client B to supply the desired data; and client B is waiting for the manager to process its write request. One solution is to use sequence numbers to order the outgoing messages for a particular block from the manager, so the sequence of events seen by any client is consistent with the manager's view.

5.3.2 Reduced Software Development Complexity

Several aspects of the Teapot language simplified the engineering of xFS. Teapot's continuations significantly reduced the number of states needed by xFS's protocol by combining each set of similar transient states into a single continuation state. By being more restrictive as well as more stylized than C, Teapot eliminated a source of programming errors. The domain-specific language also forced the decoupling of the coherence algorithm from other details of the system. This resulted in more modular protocol code that is well isolated from the rest of the file system. Finally, the domain-specific language encouraged software reuse by isolating features that are common to the class of problems they are designed to solve. In our case, we were able to borrow many support structures, such as message queues and state tables, from other protocols supplied with the Teapot release, further reducing complexity and chances of errors.

5.4 Teapot Shortcomings

Teapot was designed and is best suited for DSM environments in which the primitives available to protocol handler writers are limited and simple. The xFS coherence engine, on the other hand, must interact with other components of the system such as the kernel and the active message subsystem via more powerful operations like system calls and thread synchronizations. This difference in terms of the power and expressiveness of handler primitives has revealed some shortcomings of Teapot that were not apparent in its original application domain.

The first shortcoming is the lack of support for multithreading. An xFS client is heavily multithreaded to support concurrent users and react to concurrent requests from the network, but the coherence engine generated by Teapot has a large amount of global state and is difficult to make thread-safe. Transforming the resulting Teapot coherence engine into a monitor was unsuccessful, as subtle thread deadlocks occurred when different xFS threads enter the coherence engine and other xFS modules in different orders.

The second shortcoming concerns blocking operations on local nodes, which occur frequently in xFS coherence handlers. For example, when an xFS client needs to invalidate a cached file data block, it makes a system call to invalidate the data cached in the kernel. This system call might block, waiting for some other event that requires the attention of the coherence engine. Although Teapot provides good support for blocking operations waiting for remote messages, using the same mechanism to handle local blocking operations is tedious. In the above example, one must split the synchronous system call into asynchronous phases, invent a new node to represent the kernel, invent new states for the kernel node, invent new messages the kernel must accept and generate, and write to tie all these elements together. Better support for local blocking operations would have significantly eased the xFS protocol implementation.

The third shortcoming concerns users' inability to add new arguments to Teapot handlers. We were faced with the unpleasant dilemma of either modifying Teapot itself or simulating additional arguments via global variables. The former suggests a limitation of the model; the latter work around is bad software engineering and, in particular, it makes the multithreading problem worse. A more severe restriction is Teapot's lack of support for operations that affect blocks other than the block on which the current message arrives. The problem arises, for example, when servicing the read fault of one block by an xFS client requires the eviction of a different block. This is similar to the problem encountered by LCM during its merging phase.

6 Related Work

The Teapot work most closely resembles the PCS system by Uehara et al. at the University of Tokyo [26]. They described a framework for writing coherence protocols for distributed file system caching. Unlike Teapot, they use an interpreted language, thus compromising efficiency. Like Teapot, they write protocol handlers with blocking primitives and transform the program into a message-passing style. Our work differs in several aspects. Teapot's continuation semantic model is more general than PCS's, which is a message-driven interpretation of a protocol specification. PCS's application domain is less sensitive to protocol code efficiency, so they do not explore optimizations. Finally, we exploit verification technology by automatically generating an input specification for the MurΦ verification system.

Synchronous programming languages, such as ESTEREL [4] and the Statecharts formalism [14], are useful for describing reactive systems and real-time applications. The most important commonality between these programming languages and Teapot is that they all are ways of expressing complicated finite-state machines more intuitively than a *flat* automaton. They all support some mechanism for composing smaller, simpler state machines at the language level. A compiler then converts this composition into a flat automaton, which the programmer never has to deal with directly. ESTEREL supports decomposition of a larger state machine into smaller, concurrently-running state machines that communicate synchronously. Statecharts support the notions of depth and orthogonality to build large state machines out of smaller ones. Teapot manages the cross-product interaction (and the resulting state-space bloat) of *explicit* protocol states and pending events by factoring the pending events into states *implicit* in the continuations stack. Teapot shares another feature with ESTEREL and Statecharts in its support for automatic verification.

Teapot differs from synchronous languages in several respects. It does not have a notion of time, so it is not suitable for programming real-time applications. The notion of concurrency in synchronous languages is also different from that in Teapot. In synchronous languages, logical concurrency of state machines is convenient for expressing interacting sub-components; such concurrency is later compiled away to obtain a single-thread program. A Teapot program logically specifies only one state machine. The need for concurrency arises because several such programs are required to run on the same processing resource—they have to interleave their execution (essentially as coroutines).

Wing et al. [28] present an eloquent case for using model checking technology with complex software systems, such as a distributed file system coherence protocols. We also use model checking technology, but our primary focus is on a language for writing coherence protocols, and on deriving executable code as well as the verification system input from a single source. They write the input to the model checker separately from their code, which introduces the possibility of errors.

The design and implementation of domain-specific languages has spurred considerable interest in the systems programming community. Recent work includes instruction-set description languages [3,23], a specification language for automatically generating network packet filters [22], and compiler optimizations for interface description languages [11].

7 Conclusion: Implications for Domain-Specific Languages for Systems Software

It would be gratuitous to reiterate the successes and shortcomings of Teapot. Instead, we present some generalized insight gained from using Teapot. Although our experience is with one domain-specific language, we hope that our observations will be useful for designers of other domain-specific languages, particularly for systems software.

7.1 How big to make the language?

An important consideration when designing a domain-specific language is: how general should the language be? Teapot leans heavily to a minimal language and relies on externally-written routines. For example, it has to call a function `SameNode` to compare two values of the type `NODE`, because we could not decide how far, if at all, we wanted to support equality on opaque types in the language. Another example is whether procedure calls should be a part of the language? If so, are there any restrictions to be observed in the code for the procedures? For example, Teapot does not allow `Suspend` inside called procedures.

More comprehensive languages have the advantage that less code needs to be written in external routines. However, a larger language is harder to learn, harder to implement fully, and could be harder to optimize. While smallness has virtues, a designer should not go overboard and apply senseless restrictions. In Teapot, for example, most users were unhappy about the fixed set of arguments that appeared as handler parameters.

Capturing the commonly-occurring programming scenarios is an important role of domain-specific languages. Teapot, for example, incorporates carefully designed abstractions for waiting for asynchronous messages. However, these abstractions were less effective at capturing the scenario of waiting for asynchronous *events* in general. This kind of waiting in xFS had to be cast into the waiting-for-messages idiom using extra messages. In hindsight, the language could have been designed to support asynchronous events, with messages as a special case of events.

For problem domains where it makes sense, it is imperative to think about automatic verification from the very beginning. In Teapot, for example, we maintained a clear distinction between opaque types and their implementation. In fact, the language has no mechanism to describe the implementation of opaque types. This was done so the verification system and C code could provide an implementation suitable for their purpose, rather than providing a common base implementation which may be poor for both purposes. An example of such an abstract type is a list of sharers, which is implemented using low-level bit manipulation in C, but using an array of enumerated type 0..1 in MurΦ. The cost of this approach is that a programmer (not compiler writer) must supply the implementations, which, fortunately, are reusable.

7.2 Compiler issues

Ideally, language users should only need to know the language definition, not the details of the language implementation. Even popular general-purpose languages fall short of this ideal by great distances, at least for systems software. We have three observations in this regard. First, a language's storage allocation policy should be made clear—programmers generally like to know where in memory particular variables live and what their lifetime is. In Teapot, the storage for state parameters was not clearly defined. It was also not clear to programmers how the memory management of continuation records happened. In fact, in the current implementation, unless `Suspends` and `Resumes` dynamically match, continuation records leak, as we do not provide garbage collection. Fortunately, most protocols naturally have such balanced `Suspend` and `Resume` paths.

Second, compiler optimizations should be explicitly specified and should be under user control. Even with all the virtues of verification, a systems programmer may need low-level debuggers (perhaps for reasons unrelated to the coherence protocol). A restructuring compiler such as Teapot's makes the generated code harder to trace at runtime. Finally, despite these complications, we believe that aggressive optimizations are essential. In our experience, users are unwilling to compromise efficiency for ease of programming, particularly considering that speed is often the main purpose for distributing a computation.

7.3 Threads

As thread programming becomes commonplace, domain-specific language designers must pay close attention to thread support. Even when the language does not currently support threads, if it is successful, sooner or later users will want multithreading support. The DSL designer, due to her unique knowledge of the internals, should be prepared to provide recommendations, if not a full implementation, of thread support.

The first observation from our experience is that thread support cannot be treated as an afterthought; instead it must be an integral part of the early language design. When we attempted to make Teapot thread-safe as an add-on, we quickly discovered that global state made this an error-prone process. Even though we only introduced a small number of coarse grain locks, they frequently led to subtle synchronization problems because these locks were not exposed at the interface level. They broke abstractions and could easily lead to deadlocks. The second observation concerns the different alternatives that can enable a module written in a domain-specific language to interact with other multithreaded components. We have found that a viable alternative to making Teapot thread-safe is to turn the generated code into a single threaded *event loop* [21]. Instead of allowing multiple threads to execute concurrently in the cache coherence state machine, these threads interact with the single thread of the state machine via events. This approach eliminates unnecessary thread synchronizations inside the state machine.

7.4 Distribution and cost of entry

Most users are reluctant to even install a new programming language, much less learn it. Thus, designers of domain-specific languages should be prepared for considerable hand-holding: provide a very complete set of examples, documentation, and a distribution that builds out-of-the-box. The xFS group found that a set of complete examples was a crucial aid to adopting Teapot. However, Teapot faced two stumbling blocks: we asked our users to go pick up SML/NJ compiler from Bell Laboratories, and the MurΦ system from Stanford. Many people quit at this point, even when we offered to lead them through obstacles. Perhaps clever *perl* scripts could pick up the right software from web. Adding to our difficulties, all pieces of our system—SML compiler, MurΦ compiler, and the Teapot source—were constantly in flux, and it was very difficult to maintain coherence. We see no easy way out of this situation. From the point of view of distribution, it would be best to provide everything in portable C code. However, without drawing upon previously distributed software, we could not have built Teapot in a reasonable amount of time.

7.5 A spade is not a general-purpose earth-shattering device

A tool-builder should be up front about what a tool does and does not do. Despite our efforts, several people thought of Teapot as a verification system, which it is not. In fact, we got an inquiry about Teapot which implied that we have discovered a more practical way of doing model-checking than brute-force state-space exploration! Also, we note that Teapot is not directly suitable for describing hardware cache-coherence controllers because it permits unbounded levels of continuations. We were also asked why Teapot would not be suitable for model-checking systems unrelated to cache-coherence. These observations became apparent when people forced us to think beyond the context of Blizzard style DSMs. One should think carefully about a language's or system's restrictions and why they exist from the beginning, so as not to unnecessarily frustrate potential users.

Finally, we hope our work provides further and concrete evidence that it is better to build application-specific tools than to program complex systems with ad-hoc code. In our experience, it is more profitable to start with a focused domain-specific language or tool that solves a very specific problem to the satisfaction of a small user-community. Language extension and attempts at generalizing the application-domain should be considered only afterwards. Languages and tools with a large scope to begin with run the risk of being useful to no one, because they take much longer to design and implement, and ultimately be less useful to users than a more focused tool.

Availability

Teapot is freely distributed. Please see the Teapot page for the latest version: `http://www.cs.wisc.edu/~chandra/teapot/index.html`, or contact one of the authors.

Acknowledgments

Mark Hill brought together the xFS and the Teapot teams. Eric Eide, John McCorquodale, and the anonymous reviewers helped improve our presentation through their insightful comments.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, an NSF NYI Award CCR-9357779, and NSF Grant MIP-9625558. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[3] Mark W. Bailey and Jack W. Davidson. A Formal Model of Procedure Calling Conventions. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, California, January 1995.

[4] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. Technical Report 842, Ecole Nationale Sup'erieure des Mines de Paris, 1988.

[5] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[7] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[8] D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 44–57, December 1993.

[9] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[10] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[11] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[12] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, November 1996.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[14] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[15] Kirk L. Johnson, M. Frank Kaashoek, and Deborah A. Wallach. CRL: High Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

[16] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, October 1994.

[17] James R. Larus, Brad Richards, and Guhan Viswanathan. Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 8, pages 297–342. MITP, 1996.

[18] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[19] Chengjie Liu and Pei Cao. Maintaining Strong Cache Consistency for the World-Wide Web. Technical report, Department of Computer Science, University of Washington, May 1997.

[20] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[21] J. K. Ousterhout. Why Threads Are a Bad Idea. http://-www.sunlabs.com-/verb+ +ouster-/, 1995.

[22] Todd A. Proebsting and Scott A. Watterson. Filter Fusion. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[23] Norman Ramsey and Mary F. Fernandez. The New Jersey Machine-Code Toolkit. In *1995 Usenix Technical Conference*, pages 289–302, New Orleans, LA, January 1995.

[24] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[25] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[26] Keiraro Uehara, Hajime Miyazawa, Kouhei Yamamoto, Shigekazu Inohara, and Takasha Masuda. A Framework for Customizing Coherence Protocols of Distributed File Caches in Lucas File System. Technical Report 94-14, Department of Information Science, University of Tokyo, December 1994.

[27] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[28] Jeannette M. Wing and Mandana Vaziri-Farahani. Model Checking Software Systems: A Case Study. In *Proceedings ACM SIGSOFT Symposium On The Foundations Of Software Engineering*, October 1995.

Lightweight Languages as Software Engineering Tools

Diomidis Spinellis

University of the Aegean
83200 Karlovassi, Samos
Greece

dspin@aegean.gr

V. Guruprasad

IBM T. J. Watson Research Center
Yorktown Heights,
NY 10598, USA

prasad@watson.ibm.com

Abstract

Software subsystems can often be designed and implemented in a clear, succinct, and aesthetically pleasing way using specialized linguistic formalisms. In cases where such a formalism is incompatible with the principal language of implementation, we have devised specialized lightweight languages. Such cases include the use of repeated program code or data, the specification of complex constants, the support of a complicated development process, the implementation of systems not directly supported by the development environment, multiparadigm programming, the encapsulation of system level design, and other complex programming situations. We describe applications and subsystems that were implemented using this approach in the areas of user interface specification, software development process support, text processing, and language implementation. Finally, we analyze a number of implementation techniques for lightweight languages based on the merciless exploitation of existing language processors and tools, and the careful choice of their syntax and semantics.

1 Introduction

Software subsystems can often be designed and implemented in a clear, succinct, and aesthetically pleasing way using specialized linguistic formalisms. In cases where such a formalism is incompatible with the principal language of implementation, we have devised specialized lightweight languages. The soundness of this approach is amply demonstrated by the use of declarative languages for specialized applications [Hug90, Mos91], the attention given to very high level languages [Use94], and multiparadigm programming research [Hai86, SDE94]. The optimum formalism for implementing a subsystem is often incompatible with the language used for the rest of the system. In such cases, we propose the use of specialized lightweight languages, designed to closely match the formalism of the problem and to be easy to implement. In this way, the linguistic distance between the specification of the problem and the implementation of its solution can be minimized, resulting in cost reductions and improved project quality. In the following sections, we analyze our method's application domain, provide a number of representative case studies, outline the techniques we use, and list its advantages and associated problem areas.

2 Application Domain and Related Techniques

Lightweight languages are particularly helpful in cases where a classical implementation would require:

Repetitive sequences of code or data In many cases, one observes very similar sequences of code repeated within the program without the possibility of avoiding the repetition by the use of procedures or macros due to limitations of the programming language. In this case, the repetitive structure can be parametrically defined as a program in a specialized language. The language's compiler translates with suitable parameter substitutions the necessary parts. In [BH95], a specialized language is used to create procedures for the Ingres relational database parameterized for a given table and in [Spi93a] parameterized C code is used for the semi-automatic implementation of Haskell library functions. Section 3.3.2 describes separately compiled and interpreted filters

Specification of complex constants The constants included in a program are often the result of calculations that can be performed by a specialized language based on some basic input, or can be experimentally deduced. As an example, the size and positioning of screen objects can be calculated using an imperative or declarative specification of the required positioning. The implementation of the language converts the screen description into code or variable initialization constants. In section 3.1.1, we describe a simple language for defining GUI property boxes.

Communication with differing subsystems When the implemented system must communicate with subsystems that are based on different programming paradigms or technologies (e.g. PLCs, Postscript printers, MIDI instruments, SQL servers) a specialized language can be used as *middleware* to bridge the mismatch between the two parts. In section 3.3.1, we describe the implementation of a functional Postscript interpreter used for the communication between a relational database and a Postscript-based video titler. In section 3.4.2, we describe another Postscript-like language developed as a processing engine for CNCs.

In addition to the above application domains, lightweight languages can be beneficially employed to support:

The software development process When developing a large system, some parts of the process may not be directly supported by the development environment. Examples are the control of additional tools not covered by the system's development environment, cross-platform configuration options, and the semi-automatic production of documentation. Using specialized languages, these processes can be automated in a simple and concise way. These languages can be compiled into input for another tool (e.g. a series of *makefiles*), or they can be directly executed (e.g. to create test vectors). In section 3.2.1 we describe a simple language for defining the distribution layout of a software product using regular expression pattern matching, and in section 3.2.2, a hand crafted language for defining a project build process.

Multiparadigm programming Often a subsystem can be implemented using a different paradigm from the main application programming paradigm (e.g. functional, logic programming, CSP-based). In this case, it is often easy to implement a little compiler or interpreter to provide exactly the features required for the realization of the specific subsystem. In section 3.4.1 we describe a rule rewrite system compiler which was *inter alia* used for the implementation of a functional and a logic programming language. In [CP85] a specialized, event-based language is described. Section 3.4.2 also describes an experimental Prolog engine implemented over a Postscript base, that allowed one to combine object oriented and logic programming paradigms.

In other cases, a system's application environment may impose unnatural restrictions on the way a user interface is implemented e.g. by requiring that parts of a given command functionality be dispersed among different event procedures, resource declarations, and icon/help text definitions. A simple language can be used to define the functionality in an organized way and automatically create the code in the format required by the application environment.

Encapsulation of System Level Design It is often the case that an application's system level design can be best expressed in a specialized language. A lightweight language can be used in this case to encapsulate the design in a compact, intuitive, and maintainable formalism.

The specialized language can be implemented as an interpreter or as a compiler. The compiler is usually implemented as code generator whose target language is the main implementation language (examples 3.1.1, 3.4.1), but it can also be implemented as a subsystem that performs the translation at runtime as in example 3.3.1. Furthermore, the language can be implemented as a built-in interpreter as in example 3.3.1 aiding the easy modification of a system's parameters at runtime. The Unix *termcap* database and the application described in section 3.3.2 are examples of implementations as data stream filters.

Building interpreters and compilers for one-time use makes it a routine skill that becomes easy to apply again and again in different ways. It also helps in developing the analytical ability for dividing and conquering problems, and for orchestrating available tools, as illustrated in [SK97] and sections 3.1.2 and 3.1.3.

3 Representative Examples

In this section we outline a number of representative systems where we utilized the described methods. The examples are divided in the areas of:

- user interface specification,
- software development process support,
- text processing,
- multiparadigm programming, and
- language implementation.

3.1 User Interface Specification

The implementation of user interfaces can benefit from the use of lightweight languages for the following reasons:

- User interfaces being close to the surface of the program receive the largest amount of modification pressure. The specification of the interface in a specialized language makes such changes easier.
- Modern user interface design guidelines require the realization of modeless environments. However, structured programming is best suited for implementing modal environments. This clash can be lifted by using a lightweight language as a bridge between the two design philosophies.
- User interfaces gather input from an ordered set of command generators such as menus, toolbars, and dialog boxes, and scatter it to various processing modules. This gather/scatter operation can be easily described using a lightweight language.

In the following sections we provide some representative examples of user interface implementations that benefited from the use of lightweight languages.

3.1.1 CAD User Interface

When implementing a large CAD program [Spi93b] we faced the difficulty of organizing and maintaining all user-interface elements of the system in an productive and coherent way. The system currently comprises:

- 30 distinct drawable entities (such as line, or text),
- 40 selectable visible layers,
- 655 user prompts,
- 457 entity properties (such as the color or width of a line),
- 103 GUI string resources,
- 130 global commands,
- 98 types of tabular data, and
- 428 entity commands.

All the above are associated with help text and therefore need to be specified in more than one language. The system implementation, maintenance, and evolution was greatly simplified by specifying the above elements and their associations using lightweight languages. Some of the languages were very simple (the specification of the user prompts involves only the definition of the prompt code and the text for each language) while others were moderately complex. In all cases however the gap between the special purpose language and the resulting C code was larger than what could be effectively bridged by the use of data driven code, C macros, and encapsulation. Table 1 contains a summary of the system parts that were specified using a lightweight language, the source code lines in that language, the compiler size (in lines of Perl [WS90] code), and the resultant C++ code. The compilation from the specialized language source code to C++ resulted in code increases ranging from 44% to 657% with the mean increase being 284%. Of the total project size of about 135,000 lines, more than 37,000 were automatically generated. A representative example of the implementation style is outlined below.

Task	Source (lines)	Compiler (lines)	Output (lines)
GUI Strings	468	67	771
User Messages	2490	65	4345
Layer Control	91	244	711
Table Contents	266	618	1901
Toolbars	373	187	1568
Menus	42	81	401
Serialization	1174	297	6191
Commands	1214	248	7367
Properties	1093	839	14632
Total	7211	2646	37887

Table 1: CAD user interface specification: source, compiler, and generated code

One of the specifications for the system called for a property dialog box similar to the one found in the Delphi, MS-Access, and Visual Basic programming environments. The system supports 457 properties divided into 30 groups. Every property was given a property type such as number, angle, color, menu selection, yes/no, dialog box, and file name. We then formulated the properties in the language as illustrated in the example in Figure 1. A compiler translates the property definitions into three C code files. One of the files contains the variable and procedure definitions, the other, the procedures for initializing the supporting data structures, and the third one contains code for displaying the properties.

The compiler consists of 839 lines of Perl code and produces 14632 lines of C code. Before the compilation, the source code is passed through the C language preprocessor allowing the use of macro instructions at a minimal cost.

The description file is compiled into C in a single pass by writing the declarations and the code definitions into two distinct files; the definition file includes the declara-

```
// Standardized selections
#define ANGLE 0;45;90;135;180;225;270;315
#define TSIZE 6;8;10;12;14;16;20;24

// Text object property selection
#ifdef TEXT
//Type Caption Variable    Range  Fmt  Sel
menu  :Text    :prop_text
double:Size    :cb->size    :0 :100:%.3f:TSIZE
angle :Angle   :cb->theta   :0 :90 :     :ANGLE
bool  :Enhance:cb->enhance:     :
color :Color   :cb->color  :     :
separator
selres:Font    :cb->font    :3     :font_dir
sel   :Allign :cb->allign :Left;Right;Center
dialog:More...:CTextDialog:textdlg
#endif
```

Figure 1: Specification of a property dialog box

tion file using the C #include mechanism. In implementing the above example we made extensive use of Perl's variable substitution facility for creating customized and efficient C code.

3.1.2 Voice Shell

In a transportation management application, a voice shell (*vsh*) was developed to interface incoming telephone calls to an online database, prompted by the highly repetitive nature of code in a previously existing application. *Vsh* incorporated a *termcap*-like stack, arithmetic, voice prompts, keystroke data entry, shell escapes, and submenu invocation control, compactly representing the logic of a complete voice response system comprising about 75 speech files and 50 database access scripts within a 24x80 ASCII screen with ample space for comments. Not only was development time reduced for similar applications, but, in addition, the database accesses were simplified to short lightweight scripts. To compare, prior versions of the application would each contain 10000 or more lines of C with deeply nested "if"s, and would be generally inflexible and difficult to debug.

3.1.3 Web-based System

Reduction to lightweight languages and scripts is again a key aspect of the Papyrus online conference paper review system [Gur96]. The system comprises mostly *ksh* scripts; entire subsystems can be easily customized, thrown away or replaced. The flexibility was critical in the evolution of the package with the live requirements of an actual conference (PACT'96). The system has been or is in the process of being adopted for at least two other international conferences this year. Language components include review questionnaires parsed and evaluated using *awk* and author response form letters generated using *nroff* to format the email text.

3.2 Software Development Process Support

The software development process is a less mature field than programming in the small. The requirements, organization techniques, procedures, and development tools vary widely between organizations and projects. Lightweight languages can provide support for the above by automating repetitive work, organizing complicated tasks such as configuration control, and forcing ill-defined processes onto a formal specification vehicle. The following paragraphs outline two representative examples in the area of software development process support.

3.2.1 File Disk Layout

In a large application that we developed we had to specify the ordering of 530 files in the installation disks as well as the associated parameters. Some of the files were needed only during the setup stage and should be placed in the first disk, some of them should, additionally, not be compressed. Other files had to be installed into specific system directories while others had to be installed only as an installation option. This layout is typically specified by a GUI tool provided as part of the Microsoft Windows software development kit. The tool then distributes the files to disks and creates the installation driver program. In our case, the GUI tool could not handle the number of files needed by our application, and, in addition, had only limited grouping capabilities. We thus implemented a little language based on regular expressions that defines file sets and their installation specifications together with some global settings. Part of this specification can be seen in Figure 2.

The specification file is then translated into a file compatible with the one created by a GUI tool. In our case, the regular expression-based specification file is 66 lines long and creates a 560 line specification. The short length of our specification makes it comprehensible, readable, and ammendable to processing by other tools.

A 55 line Perl program handles the translation, identifies simple mistakes, and prints warnings for the files that fall into the default, the most general specification case, so that the user can verify that the files are given valid installation specifications.

```
# Anchor =
/distrib/
# The following are copied verbatim
=SRC = \distrib
=WRITEABLE = 1
=DISKLABEL = Application Disk 1
# Installation files (do not compress)
>DISK1,!COMPRESS,!VERSION,!READONLY,
    !DECOMPRESS,SECTION=Setup,.
+setup.exe
+setup.lst
# Library files
>ANYDISK,COMPRESS,!VERSION,!READONLY,
    DECOMPRESS,SECTION=LibFiles,.
+lib/*
# All other files (application files)
>ANYDISK,COMPRESS,!VERSION,!READONLY,
    DECOMPRESS,SECTION=AppFiles,.
+*.exe
+helplib.dll
+*
```

Figure 2: Installation file layout specification

3.2.2 Generation of Makefiles

A very simple rule-based interpreter called *archmake* provided such effective documentation and control of the build process for a 1986 project involving a few hundred source files, that it was quickly adopted as the self-documenting in-house build tool for several other multi-programmer projects before CASE tools became available. Unlike *make* [Fel79] and *imake*, the interpreter had no built-in expertise, but provided a concise and easy-to-read representation of the target object name, the component source list, the dependency generation command script, and the unit compilation script, in the following form:

```
%S description of step
# list component files to work on
%L a.c b.c c.c ...
%D mkdepend.sh %s
...
%M $(CC) -c $@ $*.c
...
```

3.3 Text Processing

As processor speeds increase, text is increasingly used as a common communication format between different subsystem parts in a quest for portability and maintainability. However, most compiled programming languages lack functionality for dealing efficiently and intuitively with text strings. The design of a small lightweight language to be embedded in a system can be used to ameliorate this deficiency by providing an efficient implementation of the facilities needed for using the specific textual interface. Two representative examples are described in the following paragraphs.

3.3.1 Video Typesetting

A statistical data relational database program we developed had to interface to a specialized video typesetting device. The device was controlled using Postscript [Inc85]. Structuring all the resulting code around Postscript was not feasible as Postscript is a relatively low-level language. Embedding large blocks of Postscript into the program was inelegant. We therefore defined a Postscript variant as a lightweight language that allowed for the embedding of meta-variable names. These names (all starting with the $ sign) were substituted at the runtime interpretation phase with the SQL query results. In order to allow for the definition of tabular data with variables of the same name the semantics of the language mimic *linear logic* in allowing each variable substitution to happen exactly once. An external loop within the code thus calls the interpreter multiple times for the same code body, each time setting the variable names to the values of the particular table row. After the implementation, all Postscript code that was embedded within the rest of the program was easily moved into external files. When a particular page needs to be displayed the file is loaded, interpreted, and downloaded into the Postscript typesetter. Most of the Postscript code of the deployed release was generated by experienced artists using specialized graphic design tools.

3.3.2 Rendering Indian languages

The Prototext language [Gur88] was designed to provide runtime filters for working with Indian language scripts using *unmodified* text editors and word processors. Indian scripts comprise a near-isomorphic family of phonetic scripts with strict rules for the graphic composition of syllabic glyphs from vowel and consonant components, and are precursors to the simpler syllabic alphabets of the Far East. The interpreter allowed the repetitive bitmap operations and the script-specific composition rules to be efficiently and compactly implemented independently of text processing code. It also facilitated the development of the fonts and composition rules on ASCII terminals before graphic editing facilities became available, and provided slightly higher performance than the not-so-optimized C code performing the same function.

3.4 Multiparadigm Programming and Language Implementation

Either in response to a genuine need, or through the occupation of individuals with domain specific knowledge, the area of programming language implementation has traditionally been a hotbed for lightweight languages. The increasing use of multiparadigm and mixed-language programming, and the possible adoption of lightweight language tools that we advocate, increase the requirements for an efficient process of language implementation. We believe that the proliferation of languages used for implementing languages stems from the need to specify formal transformation rules for a wide variety of domains that occur in a language realization. The use of a special purpose (for mature tasks such as parsing) or a lightweight language can aid the specification — and a subsequent implementation — of tokens, grammars, type systems, tree transformations, optimizations, and machine code generators. Below we outline three representative examples from our own work.

3.4.1 Rule-based Programming

We designed and implemented *term* as a lightweight language for the realization of the *blueprint* multiparadigm environment [SDE94]. For that environment we needed to implement a functional and a logic programming language. Such languages are easily implemented using declarative rules as *meta-interpreters* in their own language [SS86, p. 150] [FH88, pp. 193–195]. This practice is unfortunately associated with serious bootstrapping and performance issues. Based on the intuition that with only a moderate increase in code size the above languages can be implemented in a language that does not support important features of functional and logic programming such as deep unification, input/output parameters, backtracking, and higher order functions, we defined *term* as a simple tuple pattern matching, rule-rewrite system. *Term* is implemented in 2300 lines of *term*, yacc, and lex as a compiler from *term* to C. It was bootstrapped using a Prolog interpreter since the syntax of *term* is very similar to that of Prolog, lacking however many of Prolog's logical features. The use of *term* as an implementation vehicle proved to be a good choice, since we were then able to implement the functional and the logic languages in just 1300 lines of structured *term* code that was then compiled into efficient C code.

3.4.2 CNC extensions to Postscript

Faced with developing a complete APT (Automatically Programmed Tools) language system for CNC applications [Kra86, CM89], we found it expedient to partition the problem into a simple 22-rule Yacc-based parser front-end to translate APT to a lightweight, Postscript-like back-end language, *4th*, with minimal syntax checking. The semantic validation and *N*-D path geometry was developed directly in *4th*, which even had a *goto* to match the Fortran-ish flavor of the APT source language. The path profiling code was later reduced to a C-based extension to the interpreter, which remained as the base engine. The *4th* core was inspired by good microprocessor design: it had a control and status word, provided traps, single-stepping and call tracing, and allowed object-oriented programming support using software traps in *4th* itself. The design achieved portability, scalability, capability for backending to CAD tools, and extendibility since builtin operator table could be easily added to. This approach allowed the reduction of our profiling algorithm to three operators, implemented in 1200 lines of C and using about 1000 lines of supporting code in *4th* for full 2-1/2 D APT geometry.

An experimental Prolog extension was also built using the dictionary objects as independent "knowledge domains", unlike most implementations of Prolog, which provide only a default namespace for the functors, and provided *4th* escapes like:

```
% 4th.logic init file
{
% escape to perform inits in true 4th
(/usr/lib/4th.init) ldsrc
% push to logic domain stack
/lathe_rules load beginlogic
}
% Note-- :n means n-th from top of stack.
=(X,X) :- { :0 :1 unify }.
```

The logic programming feature could be used, for instance, to integrate tool-specific reasoning in factory automation.

3.4.3 Haskell Implementation

Haskell [HF92] is a complete general purpose purely functional programming language. When building a Haskell system [Spi93a] we utilized a number of lightweight languages for implementing:

- an efficient character classification interface based on regular expressions used for lexical analysis,
- routines for reversing, printing, and copying arbitrary parse trees,
- the language library functions in a high level form, and
- a generic machine description interface.

Of the project's total of 17,268 lines 3,610 were automatically generated. Table 2 summarizes the system parts that

Task	Source (lines)	Compiler (lines)	Output (lines)
Character classification	44	87	332
Tree inversion	527	223	441
Tree printing	527	244	1175
Tree copy	527	206	507
Library functions	178	131	410
Machine description	604	274	745
Total	1353	1165	3610

Table 2: Haskell implementation: source, compiler, and generated code

were specified using a lightweight language, the source code lines in that language, the compiler size (lines of Perl code), and the resultant C code. It is important to note that the compilers that created the tree operation routines all used the same parse tree definition C header file as source input. In this way any changes to the parse tree were automatically reflected by new tree handling routines.

4 Implementation Techniques

Lightweight languages are by their nature ephemeral and limited in scope. The effort expended in implementing them should be less than the effort required to build the system without their use. In the following sections we will outline some implementation techniques that we have successfully used to implement the languages we described in a robust yet economical way.

Use existing tools

Language implementation is a mature field in theory and technology and all relevant knowledge and tools should be capitalized when realizing a special-purpose language. The language implementation problems, solutions, and tools are taught at most computer science courses and can be put to good use when building a system in this way. Therefore, the implementation of a specially-built, lightweight language can be a valid, realistic, and cost-effective proposition.

Many simple interpreters can be implemented using text processing tools such as awk [AKW88] and Perl [WS90]. This implementation avenue can be particularly effective if the language has been designed by following the guidelines outlined in the following four paragraphs. Interpreters and compilers for more complicated languages can always be modeled using the lex/yacc [JL87] model; often performing the translation or code generation during parsing without a need to create an intermediate syntax tree.

Combine language processors

Many of the lightweight languages that we have implemented are simply preprocessors that emit another high level language. This implementation strategy is used by *yacc*, *lex*, and the *cfront* C++ implementation. In addition, the semantic richness of the lightweight language can be increased by judicious combination of a number of language processors. In many cases we pass our lightweight language source code through the C preprocessor thus adding to the language file inclusion, macro definitions, and conditional compilation facilities.

Use the features of the target language

The compilation of lightweight languages can be made easier by translating many statements or expressions directly into statements or expressions of the target language. The compiler input language can allow for parts that are not processed by the compiler, but are passed – perhaps after some minor variable name substitutions – directly to the compiled code. Languages that are compiled into C can use the `#line` preprocessor instruction to make the C compiler errors refer to the source language and not the target language. Furthermore, one pass compilation of the lightweight language can often be achieved by directing the compiled code into two files (e.g. declarations and definitions), one of which includes the other.

Use the features of the implementation tool

One other way to increase the capabilities of the lightweight language with little implementation cost is to utilize the power of the implementation tool. As an example, some tools have meta-linguistic capabilities allowing the interpretation of their own language. In this case the lightweight language source code can include expressions or statements written in the implementation language of the interpreter. In one of the languages we implemented we allowed the inclusion of full Perl expressions since these can be easily evaluated using Perl's *eval* function.

Imitate implementation tool features

Simple but robust interfaces can be easily built even with *ksh* [Kor94]. We have used simple *ksh* loops for everything ranging from compilation scaffolds and simple Web servers to entire online services (sections 3.1.2 and 3.1.3). Including simple features like shell escapes and I/O redirection is not only convenient for Unix users, but also helps in scripting and logging production runs.

Use simple syntax and provide lexical hints

To successfully use the techniques described above it is important to design the lightweight language in such a way as to make it compatible in syntax, semantics, and form with the target language, the interpreter language, or the C preprocessor.

By keeping the lightweight language small and limited in scope it should be possible for a single person to design and implement it. By a judicious selection of syntax and semantics the language can be implemented without complicated processing. Many of the languages we have designed are line and not free-form based. They can thus be compiled one line at a time, sometimes by a simple application of pattern matching and regular expressions. Line comments are also easy to remove without complicated loops and the semantic problems associated with nested block comments. An even easier to compile form of language syntax is table-based (as the example in section 3.1.1) and can thus be easily processed by field-based tools such as *awk*. The processing of variables can often be simplified be prefixing them with a special character such as $.

Make source files self-documenting

Systems we have implemented made use of as many as 13 different languages. One cannot expect a person maintaining the code to be familiar with all of them. For this reason we try to make every source file self-documenting. We try to make the language similar to well-known languages and keep its syntax and semantics simple and intuitive. A comment at the beginning of each source file of no more than a dozen lines should be all that is needed to describe the language and its use. If this is not possible, then something may be wrong with the language design.

Reprocess existing source code

The source code of a project, when written following certain conventions, can be used as as an input language for a tool that will provide additional functionality. We have thus successfully parsed header files to create a localization message database, tree processing functions (section 3.4.3), and, a variable load / save facility. In another application we have embedded error messages, explanation text, and recovery actions as comments in the source code that was processed in a later stage to create an external error message database.

5 Problems

Project architects contemplating the use of lightweight languages should carefully weigh the advantages outlined in the previous sections against a number of potential problems.

The software process covering the use of lightweight languages is not yet mature. Issues of lightweight language design in the areas of granularity, usability, interfacing, and architecture need to be examined and evaluated. The potential scalability limits of this approach should also be taken into consideration. We have used lightweight languages in projects composed of up to 200K lines of code without experiencing significant problems. Significantly larger projects may uncover problems in the areas of namespace pollution, tool portability, performance, group coordination, and resource management.

Furthermore, the developers and users of lightweight languages have to support their language on their own. Tools such as debuggers, metric analyzers, profilers, class browsers, and context-sensitive editors will be not be available. Users will have to do without them, handcraft their own, or resort to the lower level facilities provided by the underlying language. Other resources commonly available for mainstream languages such as trained developers, courses, libraries, books, and external consultants will also be absent. In addition, developers may need to reinvent techniques used for formal program validation and analysis, and re-establish metrics associated with the development process.

6 Related Work

The problem of choosing an application's implementation language forms part of language design research [Wir74, Wex76, Hoa83]. In this section we examine some representative specialized languages that justify our approach. The implementation of a number of specialized languages can be found in [AKW88] and [Ben88, pp. 83–100]. A data driven approach to structure programs is recommended in [KP78, p. 67]: "let the data structure the program." An example of a language with a narrow, specialized application domain is the one described in [CP85]. Specialized languages and tools for compiler construction are given in [JL87, Fra89, Spi93a]. Furthermore, specialized language dialects can be implemented using the system described by [CHP88].

The Unix operating system follows the tradition of using a specialized language for the definition of an application's startup status. Some of these languages are sufficiently advanced so that part of an application can often be implemented in them. As an example the *sendmail* mail transfer agent only implements a simple translation engine configured by a special startup file. All rules for routing and rewriting message headers are defined in that external configuration file (the infamous *sendmail.cf* file). Simpler yet useful examples of specialized languages in the Unix environment are the system state initialization file *inittab*, the terminal capability descriptions in *termcap*, the process schedule specification *crontab*, and the Internet daemon master switchboard *inetd*. The plethora of command initialization files for tools such as the X-Window system (*.xinitrc*), the ex editor (*.exrc*), the window manager (*.twmrc*), and the mail user agent (*.mailrc*) prompted the development of TCL [Ous94], an embedable, interpreted, string processing language aiming to provide a single consistent tool programming interface across all different tools. Finally, the *troff* [Oss82] family of Unix-based text processing tools, builds on small specialized languages for typesetting equations (*eqn* [KC74]), tables (*tbl* [Les82]), pictures (*pic* [Ker84]), and graphs (*grap* [BK86]).

Our approach builds on the ideas mentioned above by promoting the use of specialized lightweight languages as an integral part of the software development process.

7 Conclusions

The use of lightweight languages as software engineering tools bridges the semantic gap between the specification and the implementation, can offer economies of scale when implementing repetitive concepts, and can result in code that is readable, compact, easy to maintain, and concisely documents the overall structure of the application.

However, the proliferation of different languages within a project can contribute problems related to the poorly understood process, lack of tools, and scarcity of related resources. All these elements need further examination and research. We strongly believe that this research will extend the applicability of our approach and amplify the advantages brought by the use of lightweight languages in the software development process.

References

[AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[Ben88] Jon Louis Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1988.

[BH95] Olin Bray and Michael M. Hess. Reengineering a configuration-management system. *IEEE Software*, 12(1):55–63, January 1995.

[BK86] Jon Louis Bentley and Brian W. Kernighan. GRAP — a language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, August 1986.

[CHP88] James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings IEEE 1988 International Conference on Computer Languages*, Miami, USA, October 1988. IEEE Computing Society.

[CM89] Chao-Hwa Chang and Michael A. Melkanoff. *NC Machine Programming and Software Design*. Prentice-Hall, 1989.

[CP85] Luca Cardelli and Rob Pike. Squeak: a language for communicating with mice. *Computer Graphics*, 19(3):199–204, July 1985. SIGGRAPH '85 Conference Proceedings, July 22–26, San Francisco, California.

[Fel79] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, 1979.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[Fra89] Christopher W. Fraser. A language for writing code generators. *ACM SIGPLAN Notices*, 24(7):238–245, July 1989.

[Gur88] V. Guruprasad. Prototext: Universal text drivers. In *Summer 1988 Usenix Conference*, pages 331–338, San Francisco, CA, USA, June 1988. Usenix Association.

[Gur96] V. Guruprasad. Papyrus: an online paper reviewing system. http://www.pact96.ibm.com, 1996.

[Hai86] Brent Hailpern. Multiparadigm research: A survey of nine projects. *IEEE Software*, 3(1):70–77, January 1986.

[HF92] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Haskell Special Issue.

[Hoa83] C. A. R. Hoare. Hints on programming language design. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, pages 31–40. Computer Science Press, 1983. Reprinted from Sigact/Sigplan Symposium on Principles of Programming Languages, October 1973.

[Hug90] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17–42. Addison-Wesley, 1990. Also appeared in the April 1989 issue of The Computer Journal.

[Inc85] Adobe Systems Incorporated. *Postscript Language Reference Manual*. Addison-Wesley, 1985.

[JL87] Stephen C. Johnson and Michael E. Lesk. Language development tools. *Bell System Technical Journal*, 56(6):2155–2176, July-August 1987.

[KC74] Brian W. Kernighan and L. L. Cherry. A system for typesetting mathematics. Computer Science Technical Report 17, Bell Laboratories, Murray Hill, NJ, USA, May 1974. Revised April 1977.

[Ker84] Brian W. Kernighan. PIC — a graphics language for typesetting: Revised user manual. Computer Science Technical Report 116, Bell Laboratories, Murray Hill, NJ, USA, December 1984.

[Kor94] David G. Korn. Ksh - an extensible high level language. In *Very High Level Languages Symposium (VHLL)*, pages 129–146, Santa Fe, NM, USA, October 1994. Usenix Association.

[KP78] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, second edition, 1978.

[Kra86] Irving Kral. *Numerical Control Programming in APT*. Prentice-Hall, 1986.

[Les82] Michael E. Lesk. TBL — a program to format tables. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 157–174. Holt, Rinehart and Winston, seventh edition, 1982.

[Mos91] Chris Moss. Commercial applications of large Prolog knowledge bases. In H. Boley and M. M. Richter, editors, *Processing Declarative Knowledge: International Workshop PDK '91 Proceedings*, pages 32–40, Kaiserslautern, Germany, July 1991. Springer-Verlag. Lecture Notes in Computer Science 567.

[Oss82] J. F. Ossanna. NROFF/TROFF user's manual. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 196–229. Holt, Rinehart and Winston, seventh edition, 1982.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[SDE94] Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach. Language and architecture paradigms as object classes: A unified approach towards multiparadigm programming. In Jürg Gutknecht, editor, *Programming Languages and System Architectures International Conference*, pages 191–207, Zurich, Switzerland, March 1994. Springer-Verlag. Lecture Notes in Computer Science 782.

[SK97] Diomidis Spinellis and Rob Kolstad. A conversation about Perl and the shell: Choosing the implementation vehicle. *;login:*, 22(3):25–31, June 1997.

[Spi93a] Diomidis Spinellis. Implementing Haskell: Language implementation as a tool building exercise. *Structured Programming*, 14:37–48, 1993.

[Spi93b] Diomidis Spinellis. Tekton: A program for the composition, design, and three-dimensional view of architectural subjects. In *4th Panhellenic Informatics Conference*, volume I, pages 361–372, Patras, Greece, December 1993. Greek Computer Society. In Greek.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Use94] Usenix Association. *Very High Level Languages Workshop (VHLL)*, Santa Fe, Mexico, October 1994. Usenix Association.

[Wex76] Richard L. Wexelblat. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 331–336, San Fransisco, CA, USA, October 1976. IEEE Computer Society Press.

[Wir74] Niklaus Wirth. On the design of programming languages. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of IFIP Congress 74*, pages 386–393, Stockholm, Sweden, August 1974. International Federation for Information Processing, North-Holland Publishing Company.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, 1990.

A Slicing-Based Approach for Locating Type Errors

T. B. Dinesh

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
dinesh@cwi.nl

Frank Tip

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA
tip@watson.ibm.com

Abstract

The effectiveness of a type checking tool strongly depends on the accuracy of the positional information that is associated with type errors. We present an approach where the location associated with an error message e is defined as a slice P_e of the program P being type checked. We show that this approach yields highly accurate positional information: P_e is a program that contains precisely those program constructs in P that caused error e. Semantically, we have the interesting property that type checking P_e is guaranteed to produce the same error e. Our approach is completely language-independent, and has been implemented for a significant subset of Pascal.

1 Introduction

Type checkers are tools for determining the constructs in a program that do not conform to a language's type system. Type checkers are usually incorporated in interactive programming environments where they provide programmers with rapid feedback on the nature and locations of type errors. The effectiveness of a type checker crucially depends on two factors:

- The "informativeness" of the type errors reported by the tool.
- The quality of the positional information provided for type errors.

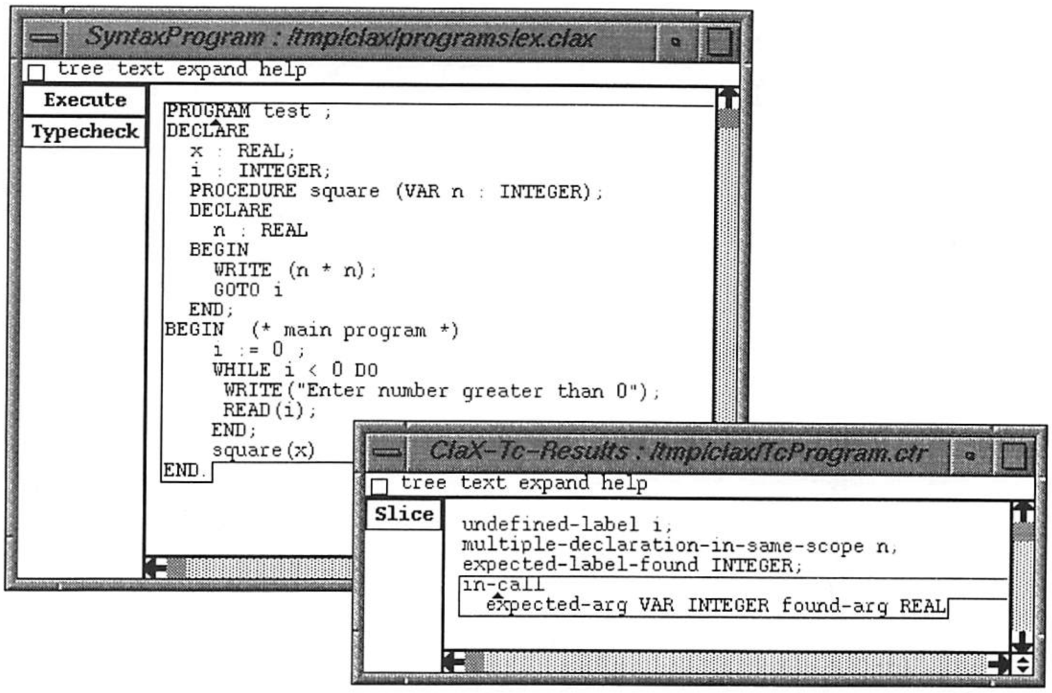
We believe that the second factor is especially important. For example, consider an assignment statement $x = y$ where x and y are of two incompatible types. What is the source of the error? Specifically, one might ask whether the assignment construct itself is "causing" the error, or if the declarations of x and y, where the incompatible types are introduced, constitute the real "source" of the error. As another example, consider a situation where a label is defined twice inside some procedure. Ideally, the location of this error would comprise *both* occurrences of the label.

We pursue a semantically well-founded approach to answer the question of what the location of a type error should be. In this approach, the behavior of a type checker is algebraically specified by way of a set of conditional equations [2], which are interpreted as a conditional term rewriting system (CTRS) [25]. These rewriting rules express the type checking process by transforming a program's abstract syntax tree (AST) into a list of error messages. We use dynamic dependence tracking [18, 28] to determine a *slice* of the original program as the positional information associated with an error message. This approach has the following advantages:

- The tracking of positional information is completely language-independent and automated; no information needs to be maintained at the specification level.

- Unlike previous approaches [12, 31], no constraints are imposed on the style in which the type checker specification is written. Error locations are always available, regardless of the specification style being used.

- The approach is semantically well-founded. If type checking a program P yields an error message e, then the location P_e associated e is a projection of P that, when type checked, will produce the same error message e. For details about semantic properties of slices, the reader is referred to [18, 28].


SyntaxProgram : /tmp/clax/programs/ex.clax
tree text expand help
Execute
Typecheck
ClaX-Tc-Results : /tmp/clax/TcProgram.ctr
tree text expand help
Slice

```
PROGRAM test ;
DECLARE
  x : REAL;
  i : INTEGER;
  PROCEDURE square (VAR n : INTEGER);
  DECLARE
    n : REAL
  BEGIN
    WRITE (n * n);
    GOTO i
  END;
BEGIN  (* main program *)
    i := 0 ;
    WHILE i < 0 DO
     WRITE("Enter number greater than 0");
     READ(i);
    END;
    square(x)
END.
```

```
undefined-label i;
multiple-declaration-in-same-scope n,
expected-label-found INTEGER;
in-call
  expected-arg VAR INTEGER found-arg REAL
```

Figure 1: The CLaX environment. The top window is a program editor with two buttons attached to it for invoking a type checker and an interpreter, respectively. The bottom window shows a list of four type errors reported by the type checker. After selecting an error message in the bottom window, the **Slice** button can be pressed to obtain the associated slice.

Although positional information is always available for any error message, the *accuracy*[1] of these locations depends on the degree to which the specified type checker's behavior is deterministic. This issue will be explored in Section 4.

We have implemented a prototype type checking system using the ASF+SDF Meta-environment [24, 31], a programming environment generator that implements algebraic specifications by way of term rewriting. Dependence tracking was previously implemented in the ASF+SDF system's term rewriting engine for the purpose of supporting dynamic slicing in generated debugging environments [29]. Figure 1 shows a snapshot of a type checking environment for the language CLaX, a Pascal-like language. The most interesting features of CLaX are: nested scopes, overloaded operators, arrays, goto statements, and procedures with reference and value parameters. The top window of Figure 1 is a program editor, which has two buttons labeled '**Type-Check**' and '**Execute**' attached to it, for invoking the type checker and the interpreter, respectively.

[1] Accuracy indicates the quality of the slice obtained. Generally, "small" slices, which contain few program constructs, are desirable because they convey the most insightful information.

The bottom window shows a list of four error messages reported by the type checker for this program.

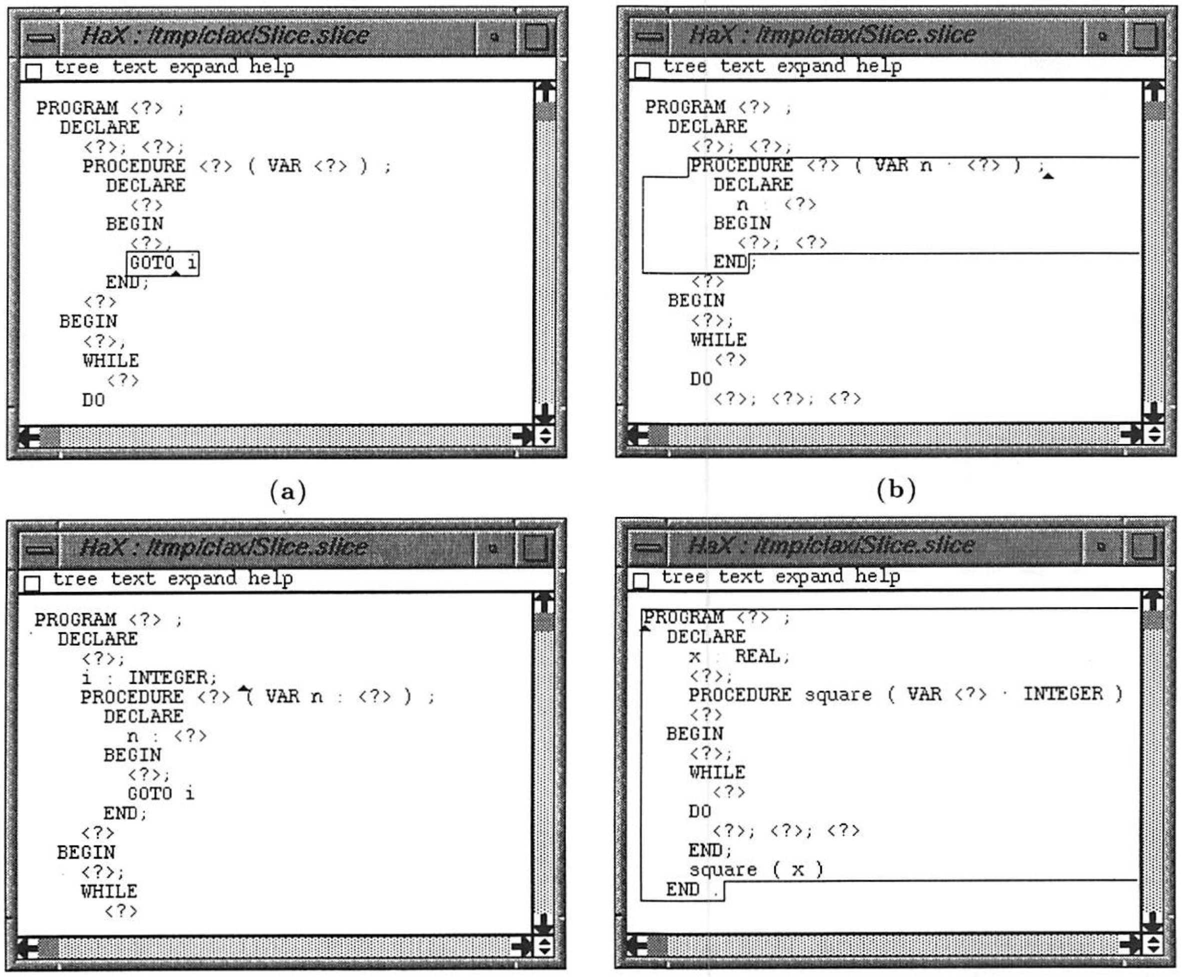
1. The first error, `undefined-label i`, indicates that the program contains a reference to a label `i`, but there is no statement with label `i` in the same scope.

2. The second error message, `multiple-declaration-in-same-scope n`, points out that an identifier `n` is declared more than once in the same scope.

3. The third error, `expected-label-found INTEGER`, indicates that the program contains an identifier that has been declared as an integer, but which is used as a label.

4. The fourth error, `in-call expected-arg VAR INTEGER found-arg REAL`, points out a type error in a procedure call. In particular, that a procedure is called with a argument type REAL when it was expecting an argument of type INTEGER.

Note that these error messages do not provide any information as to *where* the type violations occurred in the program text.



```
PROGRAM <?> ;
  DECLARE
    <?>; <?>;
    PROCEDURE <?> ( VAR <?> ) ;
      DECLARE
        <?>
      BEGIN
        <?>;
        GOTO i
      END;
    <?>
  BEGIN
    <?>;
    WHILE
      <?>
    DO
```

(a)

```
PROGRAM <?> ;
  DECLARE
    <?>; <?>;
    PROCEDURE <?> ( VAR n : <?> ) ;
      DECLARE
        n : <?>
      BEGIN
        <?>; <?>
      END;
    <?>
  BEGIN
    <?>;
    WHILE
      <?>
    DO
      <?>; <?>; <?>
```

(b)

```
PROGRAM <?> ;
  DECLARE
    <?>;
    i : INTEGER;
    PROCEDURE <?> ( VAR n : <?> ) ;
      DECLARE
        n : <?>
      BEGIN
        <?>;
        GOTO i
      END;
    <?>
  BEGIN
    <?>;
    WHILE
      <?>
```

(c)

```
PROGRAM <?> ;
  DECLARE
    x : REAL;
    <?>;
    PROCEDURE square ( VAR <?> : INTEGER )
    <?>
  BEGIN
    <?>;
    WHILE
      <?>
    DO
      <?>; <?>; <?>
    END;
    square ( x )
  END .
```

(d)

Figure 2: Slices reported by the CLaX environment for each of the type errors of Figure 1.

However, positional information may be obtained by selecting an error message and clicking on the '**Slice**' button. In Figure 2(a)–(d), the slices obtained for each of the four error messages of Figure 1 are shown[2]. Each slice is a view of the program's source indicating the program parts that contribute to the selected error. Placeholders, indicated by '<?>' in the figure, indicate program components that do not contribute to the error under consideration. The semantics of "not contributing towards a certain error message" may be characterized informally as follows: If a placeholder in the slice with respect to an error *e* is replaced with a program component of the same kind[3], type checking the resulting program is guaranteed to produce the same error *e*.

1. Figure 2(a) shows the slice for the `undefined-label` error. Clearly, the `GOTO i` statement is the source of the error, because there is no statement with label `i`.

2. Figure 2(b) shows the slice obtained for the `multiple-declaration-in-same-scope` error. The problem here is that `n` is a parameter as well as a local variable of procedure `square`. Note that both declarations of `n` occur in the slice.

3. Figure 2(c) shows the slice obtained for the `expected-label-found INTEGER` error. Note that, in addition to the `GOTO i` statement and the declaration of `i` as an `INTEGER`, all declarations in the inner scope appear in the slice. Informally, this is the case because replacing any of these declarations by declarations for variable `i` may affect the outcome of the type checking process, in the sense that the `expected-label-found INTEGER` error would no longer occur.

4. Figure 2(d) shows the slice obtained for the `in-call expected-arg VAR INTEGER found-arg REAL` error. Observe that the slice precisely indicates the program components responsible for this problem: (i) the call site `square(x)` that gave rise to the problem, (ii) the type, `INTEGER`, of `square`'s formal parameter (note that the name of this parameter is irrelevant), and (iii) the declaration of variable `x` as a `REAL`.

[2] An alternative way for displaying slices would be to highlight the corresponding text areas in the program editor of Figure 1.

[3] Although all placeholders are displayed as '<?>', placeholders are typed. In order to preserve syntactic validity of the program, an expression placeholder may only be replaced by another expression, an unlabeled-statement placeholder may only be replaced by another unlabeled-statement, etc.

The reader may observe at this point that, in addition to the program constructs responsible for a type error, a slice generally also contains certain structural information such as `BEGIN` and `END` keywords and declaration and statement list separators that are not directly related to an error. The occurrence of this structural information is due to the way slices are computed. If desired, displaying this information could easily be suppressed to a large extent. For example, removal of all `BEGIN`, `END`, and `DECLARE` keywords and list separators from the computed slices would reduce the amount of "noise" considerably. In certain cases, slices may contain `IF` or `WHILE` statements whose condition and body are omitted from the slice (see, e.g., Figure 2(d)). Such constructs can also be removed from the slice without affecting the semantic content. We consider slice postprocessing to be primarily a user-interface issue, which is outside the scope of this paper.
The remainder of the paper is organized as follows. Section 2 presents our approach for specifying type checkers. In Section 3, the use of term rewriting for executing specifications is discussed. In addition, dependence tracking, the mechanism for computing slices is presented. Section 4 is concerned with the effect of determinism in the specification on slice accuracy. In Section 5, related work is discussed. In particular, the slice notion introduced in the present paper is compared with the traditional notion of a program slice. Conclusions and possible directions for future work are stated in Section 6.

2 Specification of Static Semantics and Type Checking

A *static semantics* specification only determines the validity of a program and is not concerned with pragmatic issues such as the source location where a violation of the static semantics occurred, or even what program construct caused the violation. A *type checker* specification typically uses the static semantics specification as a guideline, and specifies the presentation and source location of type errors in invalid programs. Adding such reporting information to a static semantics specification is a cumbersome and error-prone task, because keeping track of positional information can be nontrivial, especially if multiple program fragments together constitute a type error.

[Eq1]	tc(begin *Decls Stats* end)	= dist(*Stats*, tenv(*Decls*))
[Eq2]	dist(*Stat*$_1$; *Stat*$_2$, *Tenv*)	= dist(*Stat*$_1$, *Tenv*); dist(*Stat*$_2$, *Tenv*)
[Eq3]	dist(*Id* := *Exp*, *Tenv*)	= dist(*Id*, *Tenv*) := dist(*Exp*, *Tenv*)
[Eq4]	dist(*Exp*$_1$ + *Exp*$_2$, *Tenv*)	= dist(*Exp*$_1$, *Tenv*) + dist(*Exp*$_2$, *Tenv*)
[Eq5]	dist(*Id*, *Tenv*)	= type-of(*Id*, *Tenv*)
[Eq6]	type-of(*Id*, tenv(T_1^*; *Id* : *Type*; T_2^*))	= *Type*
[Eq7]	natural + natural	= natural
[Eq8]	natural := natural	= "correct"

Figure 3: Static semantics specification for determining the validity of assignments.

In [14], we introduced an abstract interpretation style for writing static semantics specifications. In a nutshell, this style advocates the following:

- reducing program constructs to their *type*,
- evaluating type expressions at an abstract level, and
- only specifying the type-correct cases.

Operationally, the static semantics specification describes a transformation of a program to a set of type-expressions for program constructs that are type-incorrect.

Figure 3 shows a tiny static semantics specification for determining the validity of assignment statements in straight-line flow programs. The reader should be aware that this specification only serves to illustrate the general style of specifying a static semantics and is incomplete; for example, it does not verify if variables are declared more than once. Equation **[Eq1]** defines a top-level function tc for checking a program. Informally, **[Eq1]** states that checking a program involves (i) creating an initial type-environment that contains variable-type pairs, and (ii) distributing the type-environment over the program's statements, using an auxiliary function dist. For the simple example we study here, the type-environment consists of the declaration section of the program, to which the constructor function tenv is applied. Equation **[Eq2]** expresses the distribution of type-environments over lists of statements, and **[Eq3]** and **[Eq4]** the distribution over assignment operators and '+' operators, respectively. **[Eq5]** states how an identifier is reduced to its type, using an auxiliary function type-of, which is defined in **[Eq6]**. Note that the variables T_1^* and T_2^* in **[Eq6]** match any sublist of (zero or more) declarations in a declaration section. Equation **[Eq7]** expresses the abstract evaluation of additions, and **[Eq8]** states that the assignment of a natural expression to a natural variable is valid.

As an example, consider checking the following program block:

```
tc(begin x :  natural; y :  string;
   x := x + x; x := y + x end)
```

Application of **[Eq1]** results in:

```
dist(x := x + x; x := y + x,
     tenv(x :  natural; y :  string))
```

Application of **[Eq2]** yields:

```
dist(x := x + x,
     tenv(x :  natural; y :  string));
dist(x := y + x,
     tenv(x :  natural; y :  string))
```

At this point, **[Eq3]** can be applied to both components, producing:

```
dist(x, tenv(x :  natural; y :  string))
:= dist(x + x,
        tenv(x :  natural; y :  string));
dist(x, tenv(x :  natural; y :  string))
:= dist(y + x,
        tenv(x :  natural; y :  string))
```

The left-hand sides of both assignments can be reduced to their types using **[Eq5]** and **[Eq6]**, resulting in:

```
natural :=
  dist(x + x,
       tenv(x :  natural; y :  string));
natural :=
  dist(y + x,
       tenv(x :  natural; y :  string))
```

Using **[Eq4]** and **[Eq5]**, the right-hand sides of the assignments can be simplified:

```
natural := natural + natural;
natural := string + natural
```

[Er1]	msgs($Stat_1$; $Stat_2$)	= msgs($Stat_1$);msgs($Stat_2$)
[Er2]	msgs("correct")	= "No errors"
[Er3]	$Msg*$; "No errors"; $Msg*'$	= $Msg*$; $Msg*'$
[Er4]	msgs(T_1 := T_2) when simpletype(T_2) $\neq$ true	= msgs(T_2)
[Er5]	msgs(T_1 := T_2) when simpletype(T_2) = true	= "Incompatible types in assignment."
[Er6]	msgs(T_1 + T_2)	= "Operands of + should have the same type."
[Er7]	simpletype(natural)	= true
[Er8]	simpletype(string)	= true

Figure 4: Postprocessing to obtain human-readable messages.

Using equation [Eq7], the first assignment can be simplified:

```
natural := natural;
natural := string + natural
```

Finally, application of [Eq8] yields the final result:

```
"correct";
natural := string + natural
```

The fact that this term contains a subterm that cannot be reduced to "correct" indicates that the program is not type-correct. Note that the non-"correct" subterm already gives a rough indication of the nature of the type violation.
Figure 4 shows a set of equations that define a function msgs that transforms the cryptic messages produced by the specification of Figure 3 into human-readable form. The equations of Figure 4 assume that the term to which they are applied is fully normalized w.r.t. type checking equations of Figure 3. Equation [Er1] distributes function msgs over all statements in a block. [Er2] transforms the constant correct, which was derived from a type-correct program construct, into a message "No errors". Since we are not interested in generating messages for correct statements, equation [Er3] eliminates "No errors" from lists of messages. Equations [Er4] and [Er5] perform the post-processing of expressions that are derived from incorrect assignment statements. Note that these equations are *conditional*: they are only applicable if a certain condition holds. (Here, the condition verifies if the right-hand side of the expression is a simple type, using auxiliary equations [Er7] and [Er8].) [Er4] postprocesses assignment statements whose right-hand side consists of an irreducible expression; whereas [Er5] postprocesses assignments whose left-hand side and right-hand side are incompatible. Equation [Er6] postprocesses '+' expressions with incompatible arguments. The reader should observe that the specification of Figure 4 only serves to illustrate the general technique and that it is incomplete; For example, it does not handle nested expressions.
As an example, we will postprocess the term "correct"; natural := string + natural by applying the equations of Figure 4 to the term:

```
msgs("correct";
     natural := string + natural)
```

Applying [Er1] produces:

```
msgs("correct");
msgs(natural := string + natural)
```

Using equation [Er2], we obtain:

```
"No errors";
msgs(natural := string + natural)
```

By applying [Er3], the "No errors" message is eliminated:

```
msgs(natural := string + natural)
```

Since the right-hand side of the assignment is not of a simple type (we cannot derive the constant true from the term simpletype(string + natural), conditional equation [Er4] can be applied, producing:

```
msgs(string + natural)
```

Application of [Er6] yields the human readable error message:

```
"Operands of + should have the same type."
```

The CLaX type checker specification that has been used to generate the snapshots of Figures 1 and 2 follows the same basic principles that have been presented in this section. Language features such as

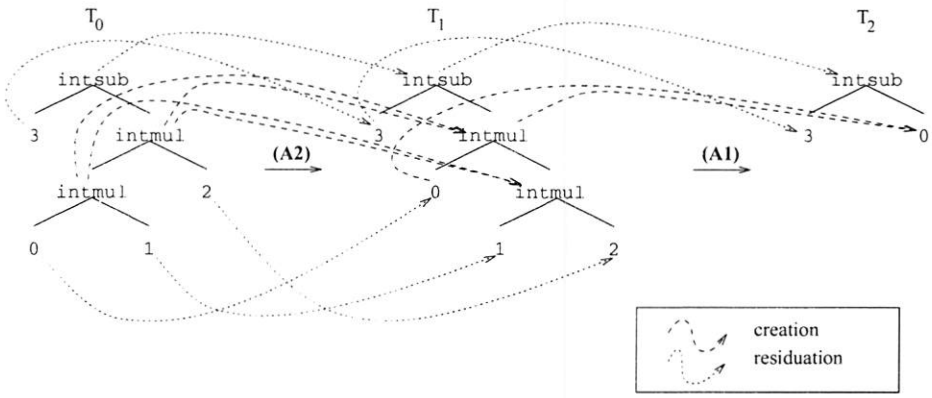


Figure 5: Example of creation and residuation relations.

gotos, nested scopes, and arrays introduce some additional complexity, but we experienced no fundamental problems. An annotated listing of the CLaX specification will appear in a technical report in the near future [15]. A previous version of the CLaX specification may be found in [14].

3 Term Rewriting and Dependence Tracking

In the previous section, specifications were "executed" by repeatedly applying equations to terms—a mechanism that is usually referred to as *term rewriting.* Both theoretical properties of term rewriting systems [25] such as termination behavior, and efficient implementations of rewriting systems [22, 23] have been studied extensively.

Term rewriting [25] can be viewed as a cyclic process where each cycle begins by determining a subterm t and a rule $l = r$ such that t and l *match.* This is the case if a substitution σ can be found that maps every variable X in l to a term $\sigma(X)$ such that $t \equiv \sigma(l)$ (σ distributes over function symbols). For rewrite rules without conditions, the cycle is completed by replacing t by the instantiated right-hand side $\sigma(r)$. A term for which no rule is applicable to any of its subterms is called a *normal form*; the process of rewriting a term to its normal form (if it exists) is referred to as *normalizing.* A conditional rewrite rule [3] (such as **[Er4]** and **[Er5]** in Figure 3) is only applicable if all its conditions succeed; this is determined by instantiating and normalizing the left-hand side and the right-hand side of each condition. Positive (equality) conditions (of the form $t_1 = t_2$) succeed iff the resulting normal forms are syntactically equal, negative (inequality) conditions ($t_1 \neq t_2$) succeed if they are syntactically different.

Thus far, we have described the process of specifying a type checker, and the execution of such specifications by way of term rewriting. In order to obtain positional information, we use a technique called *dependence tracking* that was developed by Field and Tip [18, 28]. For a given sequence of rewriting steps $T_0 \rightarrow \cdots \rightarrow T_n$, dependence tracking computes a slice of the original term, T_0, for each function symbol or subcontext (a notion that will be presented below) of the result term, T_n.

We will use the following simple specification of integer arithmetic (taken from [29]) as an example to illustrate dependence tracking:

```
[A1]  intmul(0, X)                 =  0
[A2]  intmul(intmul(X, Y), Z)      =
          intmul(X, intmul(Y, Z))
```

By applying these equations, the term `intsub(3, intmul(intmul(0, 1), 2))` may be rewritten as follows (subterms affected by rule applications are underlined):

$$T_0 = \texttt{intsub(3,}\underline{\texttt{intmul(intmul(0, 1), 2)}}\texttt{)} \xrightarrow{} \textbf{[A2]}$$

$$T_1 = \texttt{intsub(3,}\underline{\texttt{intmul(0, intmul(1, 2))}}\texttt{)} \xrightarrow{} \textbf{[A1]}$$

$$T_2 = \texttt{intsub(3,0)}$$

By carefully studying this example, one can observe the following:

- The outer context `intsub(3, ●)` of T_0 ('●' denotes a missing subterm) is not affected at all, and therefore reappears in T_1 and T_2.

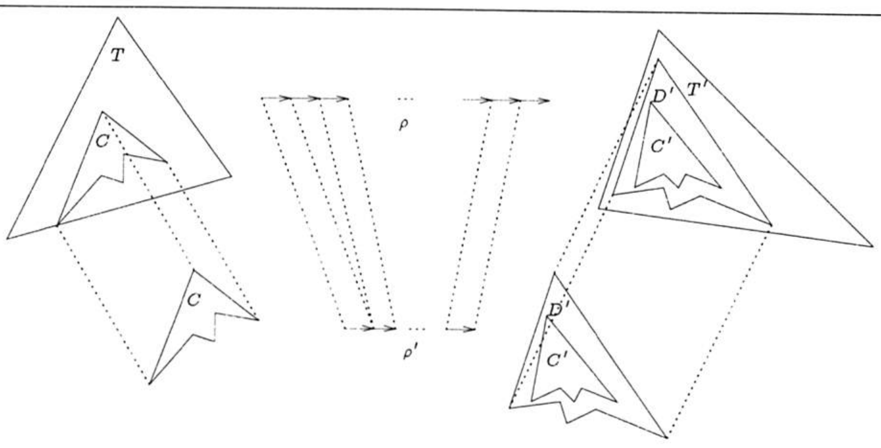


Figure 6: Depiction of the definition of a term slice.

- The occurrence of variables X, Y, and Z in both the left-hand side and the right-hand side of [A2] causes the respective subterms `0`, `1`, and `2` of the underlined subterm of T_0 to reappear in T_1.
- Variable X only occurs in the left-hand side of [A1]. Consequently, the subterm `intmul(1, 2)` (of T_1) that is matched against X does not reappear in T_2. In fact, we can make the stronger observation that the subterm matched against X is *irrelevant* for producing the constant `0` in T_2: the "creation" of this subterm `0` only requires the presence of the context `intmul(0, ●)` in T_1.

The above observations are the cornerstones of the dynamic dependence relation of [18, 28]. Notions of *creation* and *residuation* are defined for single rewrite-steps. The former involves function symbols produced by rewrite rules whereas the latter corresponds to situations where symbols are copied, erased, or not affected by rewrite rules[4]. Figure 5 shows all residuation and creation relations for the example reduction discussed above.

Roughly speaking, the dynamic dependence relation for a sequence of rewriting steps ρ consists of the transitive closure of creation and residuation relations for the individual steps in ρ. In [18, 28], the dynamic dependence relation is defined as a relation on *contexts*, i.e., connected sets of function symbols in a term. The fact that C is a *subcontext* of a term T is denoted $C \sqsubseteq T$. For any sequence of rewrite steps $\rho : T \rightarrow \cdots \rightarrow T'$, a *term slice* with respect to some $C' \sqsubseteq T'$ is defined as the subcontext $C \sqsubseteq T$ that is found by tracing back the dynamic dependence relations from C'. The term slice C satisfies the property that C can be rewritten to a term $D' \sqsupseteq C'$ via a sequence of rewrite steps ρ', where ρ' contains a subset of the rule applications in ρ. This property is illustrated in Figure 6.

Returning to the example, we can determine the term slice with respect to the entire term T_2 by tracing back all creation and residuation relations to T_0. The reader may verify that the term slice with respect to `intsub(3, 0)` consists of the context `intsub(3, intmul(intmul(0, ●), ●))`.

The bottom window of the CLaX environment of Figure 1 is a textual representation of a term that represents a list of errors. The slices shown in Figure 2(a)–(d) are computed by tracing back the dependence relations from each of the four "error" subterms.

4 The Effect of Determinism on Slice Accuracy

We have argued that our approach for obtaining positional information does not rely on a specific specification style. Nevertheless, experimentation with the CLaX type checker has revealed that the *accuracy* of the computed slices inversely depends on the degree to which the specification is *deterministic*. As a general principle, *more* determinism in a specification leads to *less* accurate slices. To understand why this is the case, consider the nature of dynamic dependence relations. Suppose that type

[4]The notions of creation and residuation become more complicated in the presence of so-called *left-nonlinear* rules and *collapse rules*. This is discussed at greater length in [18, 28].

checking a program P involves a sequence of rewrite steps r that ultimately lead to an error e. The slice P_e associated with e has the property that it can be rewritten to a term containing e, using a subset r' of the rewrite-steps in r. If the rewrite steps in r encode a deterministic process such as the explicit traversal of a list of statements, this deterministic behavior will also be exhibited by r', to the extent that it contributed to the creation of e.
As an example, consider rewriting the term:

```
type-of(tenv( x :  integer; y:  string;
              z :  integer), y)
```

according to the specification of Figure 3. By applying equation **[Eq6]**, this term rewrites to the constant `string`. By tracing back the dynamic dependence relations, we find that the context

```
type-of(tenv(●; y:  string; ●), y)
```

was needed to create this result. Now suppose that instead of equation **[Eq6]**, we use the following two equations for reducing the same term:

[Eq6a] `type-of(`*Id*`, tenv(`*Id*`:`*Type*`;` D^*`)) =`
    Type

[Eq6b] `type-of(`*Id*`, tenv(`*Id′*`:`*Type*`;` D^*`)) =`
    `type-of(`*Id*`, tenv(` D^*`))`
      `when` *Id′* `!=` *Id*

The resulting term would be the same as before: the constant `string`, which is obtained by first applying equation **[Eq6b]** followed by applying equation **[Eq6a]**. However, the subcontext needed for creating this result would now consist of:

```
type-of(tenv(x :  ●; y:  string; ●), y)
```

The variable `x` in the first element of the type environment is now included in the slice because the *order* in which the type environment is traversed is made explicit in the specification. Informally stated, the resulting term `string` is now dependent on the fact that the first element of the type environment is not an entry for variable `y`.
The use of list functions and list matching in specifications (i.e., allowing function symbols with a variable number of arguments and variables that match sublists) has the effect of reducing determinism, and therefore improving slice accuracy. We believe that more powerful mechanisms for expressing nondeterminism such as higher-order functions [21] can in principle improve slice accuracy even further.
Experimentation with the CLaX type checker specification of [14] revealed a small number of cases where slices were unnecessarily inaccurate due to overly deterministic behavior. Virtually all of these cases consisted of explicit traversals of lists, with the purpose of finding a specific list element, or verifying whether or not a list contained a certain element more than once. In each of these cases, the use of list functions allowed us to specify the same function nondeterministically with little effort. In a forthcoming technical report [15], we will present a brief overview of a few of the more interesting changes we made to the CLaX specification in order to make it less deterministic.

5 Related Work

The work presented in this paper is closely related to earlier work by the same authors. The CLaX type checker [14] was developed in the context of the COMPARE (compiler generation for parallel machines) project, which was part of the European Union's ESPRIT-II program. We originally used *origin tracking* [11] to associate source locations with type errors. Origin tracking is similar in spirit to dependence tracking in the sense that it establishes relationships between subterms of terms that occur in a rewriting process. The key difference between the two techniques is that origin tracking establishes relationships between *equal* subterms (either syntactically equal, or equal in the algebraic sense), whereas dependence tracking determines for each subterm the context needed to create it. The use of origin tracking for obtaining positional information was further investigated in [12, 13]. Although the results were encouraging (in terms of accuracy of positional information), origin tracking was found to impose restrictions on the style in which the type checker specification was written. Since origin tracking only establishes relationships between equal terms, the error messages generated by the type checker must contain fragments that literally occur in the program source; otherwise, positional information is unavailable. In [12, 13], this problem was circumvented by tokenization, i.e., using an applicative syntax structure and rewriting the specification in such a way that error messages always contain literal fragments of program source, which guarantees the non-emptyness of origins. Modification of the type checker specification resulted in adequate positional information for type errors. By contrast, our approach does not require any modifications to specifications at all. In the previous section, we have described techniques for improving the quality of positional information by avoiding determinism in specifications, but it should be emphasized that such improvements are completely optional.

The dependence tracking relation we use for obtaining positional information was developed by Field and Tip [18, 28] for the purpose of computing program slices. A *program slice* [33, 34, 30] is usually defined as the set of statements in a program P that may affect the values computed at the *slicing criterion*, a designated point of interest in P. Two kinds of program slices are usually distinguished. *Static* program slices are computed using compile-time dependence information, i.e., without making assumptions about a program's inputs. In contrast, *dynamic* program slices are computed for a specific execution of a program. An overview of program slicing techniques can be found in [30].

By applying dependence tracking to different rewriting systems, various types of slices can be obtained. In [17] programs are translated to an intermediate graph representation named PIM [16, 1]. An equational logic defines the optimization/simplification and (symbolic) execution of PIM-graphs. Both the translation to PIM and the equational logic for simplification of PIM-graphs are implemented as rewriting systems, and dependence tracking is used to obtain program slices for selected program values. By selecting different PIM-subsystems, different kinds of slices can be computed, allowing for various cost/accuracy tradeoffs to be made. In [29], dynamic program slices are obtained by applying dependence tracking to a previously written specification for a CLaX-interpreter.

The slice notion presented in the current paper differs from the traditional program slice concept in the following way. In program slicing, the objective is to find a projection of a program that preserves part of its *execution* behavior. By contrast, the slice notion we have used here is a projection of the program for which part of another program property—*type checker* behavior—is preserved. It would be interesting to investigate whether there are other abstract program properties for which a sensible slice notion exists.

Another approach to providing positional information for type errors is pursued by van Deursen [10, 9]. Van Deursen investigates a restricted class of algebraic specifications called Primitive Recursive Schemes (PRSs). In a PRS, there is an explicit distinction between constructor functions that represent language constructs, and other functions that process these constructs. Van Deursen extends the origin tracking notion of [11] by taking this additional structure into account, which enables the computation of more precise origins.

Heering [21] has experimented with higher-order algebraic specifications to specify static semantics. We believe that the approach of this paper would work very well with higher-order specifications, since these allow one to avoid deterministic behavior, which adversely affects slice accuracy. However, this would require extension of the dependence tracking notion of [18, 28] to higher-order rewriting systems.

Fraer [20] uses a variation on origin tracking [6, 5, 7] to trace the origins of assertions in a program verification system. In cases where an assertion cannot be proved, origin tracking enables one to determine the assertions and program components that contributed to the failure of the verification condition.

Flanagan et al. [19] have developed MrSpidey, an interactive debugger for Scheme, which performs a static analysis of the program to determine operations that may lead to run-time errors. In this analysis, a set of abstract values is determined for each program construct, which represents the set of run-times values that may be generated at that point. These abstract values are obtained by deriving a set of constraints from the program in a syntax-directed fashion, which approximate the data flow in the program. In addition, a value flow graph is constructed, which models the flow of values between program points. MrSpidey has an interactive user-interface that allows one to visually inspect values as well as flow-relationships.

6 Conclusions

We have presented a slicing-based approach for determining locations of type errors. Our work assumes a framework in which type checkers are specified algebraically, and executed by way of term rewriting [25]. In this model, a type check function rewrites a program's abstract syntax tree to a list of type errors. Dynamic dependence tracking [18, 28] is used to associate a *slice* [33, 30] of the program with each error message. Unlike previous approaches for automatic determination of error locations [14, 12, 13, 10, 9, 6, 5, 7], ours does not rely on a specific specification style, nor does it require additional specification-level information for tracking locations. The computed slices have an interesting semantic property: The slice P_e associated with error message e is a projection of the original program P that, when type checked, is guaranteed to produce the same type error e.

We have implemented this work in the context of the ASF+SDF Meta-environment [24, 31] for a substantial subset of Pascal. Experimentation with CLaX revealed that the computed slices provide highly insightful information regarding the nature of type violations. We have observed that the amount of determinism in a specification is an important factor that determines the accuracy of the computed slices, and we consider this to be a topic that requires further study. As another direction for future work, we intend to study the applicability of slicing-based error location in the related area of type *inference* [8], in particular for object-oriented languages [27] and for ML [26]. Providing accurate positional information for type inference errors in ML is a difficult problem. Several proposals that rely on adapting or extending the underlying type system or inference algorithm have been presented (see, e.g., [4, 32]). In contrast, we are interested in an approach that requires no changes to type inference algorithm or the type system. The basic idea is to apply dependence tracking to a rewriting-based implementation of an ML type inferencer. Although a slice can be computed for each reported type inference error, it is unclear how accurate such slices will be in practice.

References

[1] BERGSTRA, J., DINESH, T., FIELD, J., AND HEERING, J. A complete transformational toolkit for compilers. In *Proc. European Symposium on Programming* (Linköping, Sweden, April 1996), vol. 1058 of *Lecture Notes in Computer Science*, Springer-Verlag. Full version: Technical Report CS-R9646, Centrum voor Wiskunde en Informatica (CWI), Amsterdam; To appear in TOPLAS, 1997.

[2] BERGSTRA, J., HEERING, J., AND KLINT, P., Eds. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[3] BERGSTRA, J., AND KLOP, J. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences 32*, 3 (1986), 323–362.

[4] BERNSTEIN, K. L., AND STARK, E. W. Debugging type errors (full version). Tech. rep., State University of New York at Stony Brook, Computer Science Department, 1995.

[5] BERTOT, Y. Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 327–337. *SIGPLAN Notices* 26(6).

[6] BERTOT, Y. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle.* PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

[7] BERTOT, Y. Origin functions in lambda-calculus and term rewriting systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming (CAAP '92)* (1992), J.-C. Raoult, Ed., vol. 581 of *LNCS*, Springer-Verlag.

[8] CLÉMENT, D., DESPEYROUX, J., DESPEYROUX, T., AND KAHN, G. A simple applicative language: Mini-ml. In *Proc. 1986 ACM Symposium on Lisp and Functional Programming* (1986), pp. 13–27.

[9] DEURSEN, A. V. *Executable Language Definitions—Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

[10] DEURSEN, A. V. Origin tracking in primitive recursive schemes. Report CS-R9401, Centrum voor Wiskunde en Informatica (CWI), 1994.

[11] DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. *Journal of Symbolic Computation 15* (1993), 523–545.

[12] DINESH, T. B. Type checking revisited: Modular error handling. In *Semantics of Specification Languages* (1994), D. J. Andrews, J. F. Groote, and C. A. Middelburg, Eds., Workshops in Computing, Springer-Verlag, pp. 216–231. Utrecht 1993.

[13] DINESH, T. B. Typechecking with modular error handling. In *Language Prototyping: An Algebraic Specification Approach*, A. v. Deursen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co., 1996, pp. 85–104.

[14] DINESH, T. B., AND TIP, F. Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), 1992.

[15] DINESH, T. B., AND TIP, F. A slicing-based approach for locating type errors. Tech. rep., CWI/IBM, 1997. Forthcoming.

[16] FIELD, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the*

ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (1992), pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR-909.

[17] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[18] FIELD, J., AND TIP, F. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming* (1994), M. Hermenegildo and J. Penjam, Eds., vol. 844, Springer-Verlag, pp. 415–431.

[19] FLANAGAN, C., FLATT, M., KRISHNAMUTHI, S., WEIRICH, S., AND FELLEISEN, M. Catching bugs in the web of program invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, 1996), pp. 23–32.

[20] FRAER, R. Tracing the origins of verification conditions. In *Proceedings of AMAST'96* (Munich, Germany, July 1996), vol. 1101, Springer-Verlag LNCS.

[21] HEERING, J. Second-order term rewriting specification of static semantics. In *Language Prototyping: An Algebraic Specification Approach*, A. v. Deursen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co., 1996, pp. 295–306.

[22] KAMPERMAN, J. *Compilation of Term Rewriting Systems.* PhD thesis, University of Amsterdam, 1996.

[23] KAMPERMAN, J., AND WALTERS, H. Minimal term rewriting systems. In *Recent trends in data type specification : 11th workshop on specification of abstract data types joint with the 8th COMPASS workshop: Oslo, Norway, 19-23.09.1995 : selected papers* (1996), vol. 1130 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 274–290.

[24] KLINT, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology 2*, 2 (1993), 176–201.

[25] KLOP, J. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 1–116.

[26] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML.* The MIT Press, Cambridge, MA, 1990.

[27] PALSBERG, J., AND SCHWARTZBACH, M. *Object-Oriented Type Systems.* John Wiley & Sons, 1993.

[28] TIP, F. *Generation of Program Analysis Tools.* PhD thesis, University of Amsterdam, 1995.

[29] TIP, F. Generic techniques for source-level debugging and dynamic program slicing. In *Proceedings of the Sixth International Joint Conference on Theory and Practice of Software Development* (Aarhus, Denmark, May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., vol. 915 of *LNCS*, Springer-Verlag, pp. 516–530.

[30] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[31] VAN DEURSEN, A., HEERING, J., AND KLINT, P., Eds. *Language Prototyping—An Algebraic Specification Approach*, vol. 5 of *AMAST Series in Computing.* World Scientific, 1996.

[32] WAND, M. Finding the source of type errors. In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, 1986), pp. 38–43.

[33] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.

[34] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

Typed Common Intermediate Format

Zhong Shao
Dept. of Computer Science
Yale University
New Haven, CT 06520-8285
shao-zhong@cs.yale.edu

Abstract

Application languages are very effective in solving specific software problems. Unfortunately, they pose great challenges to reliable and efficient implementations. In fact, most existing application languages are slow, interpreted, and have poor interoperability with general-purpose languages.

This paper presents a framework on building high-quality systems environment for multiple advanced languages. Our key innovation is the use of a common typed intermediate language, named FLINT, to model the semantics and interactions of various language-specific features. FLINT is based on a predicative variant of the Girard-Reynolds polymorphic calculus F_ω*, extended with a very rich set of primitive types and functions.*

FLINT provides a common compiler infrastructure that can be quickly adapted to generate compilers for new general-purpose and domain-specific languages. With its single unified type system, FLINT serves as a great platform for reasoning about cross-language interoperations. FLINT types act as a glue to connect language features that complicate interoperability, such as mixed data representations, multiple function calling conventions, and different memory management protocols. In addition, because all runtime representations are determined by FLINT types, languages compiled under FLINT can share the same system-wide garbage collector and foreign function call interface.

1 Introduction

Application languages (a.k.a. domain-specific languages or DSLs) are very effective in solving specific software problems. Unfortunately, their focus on a particular domain and their (often) quick turnaround time make it unrealistic to develop full-scale compilers from scratch. In fact, due to the lack of compiler infrastructures, many existing application languages are interpreted but not compiled. As a result, software written in application languages are generally slow and have poor interoperability with general-purpose languages.

The interoperability problem, of course, also applies to advanced type safe languages such as Java [10], Modular-3 [27], ML [22], and Haskell [17]. Each of these programming languages, whether general-purpose or domain specific, often has its own syntax, semantics, and implementation specifics; it also must run under a special runtime system with its own garbage collector and foreign function call interface (to the low-level C code). Interoperation or communication among these languages is a nightmare, if not impossible. Several recently proposed object models (e.g, Microsoft's COM [32] and OMG's CORBA [11]) offer a partial solution, however, they all require that programs written in different languages run under different hardware-protection domains (i.e., address space). Wallach *et al* [41] have shown that cross-domain function calls under COM can be a factor of 1000 times slower than the regular function calls within a single domain. This is unacceptable for many applications.

The problem on (lack of) compiler infrastructures is even more serious. To write a compiler for a new language L, one has to either write everything from scratch, or compile L into some main-stream languages such as C and C++. However, for most advanced languages, C is much too low-level to serve as a good target language. Modern languages often support strong typing, automatic memory management, program exceptions, and higher-order functions (or closures), but C does not support any of these. Most C compilers are not designed to produce good code for these higher-level language features. To write a compiler from L to C, one still must write multiple compilation phases and customize her own runtime system (including support to garbage collection, proper signal-handling, and foreign-function call interface).

This paper presents a new framework on building high-quality systems environment for *multiple* general-purpose and application-oriented languages. We are particularly interested in the class of HOT[1] languages, namely, languages that are *Higher-Order* and *Typed*. With a broader interpretation, we use Higher-Order to include languages where objects contain methods (even though functions are not first-class citizens), and Typed to include both static and dynamic typing. Thus, Java is HOT, so is ML, Haskell and Scheme. Because application languages are designed to exploit a higher-level of abstraction and program analysis, many of them are designed to be HOT as well.

Our key innovation is the use of a common typed intermediate language, named FLINT, to model the semantics and interactions of various language-specific features. FLINT is based on a predicative variant of the Girard-Reynolds polymorphic lambda calculus F_ω [9, 31], extended with a very rich set of primitive types and functions. Although HOT languages can be very different in semantics, they all have a mathematically rigorous type system. The fact that almost all HOT features can be compiled into an F_ω-like calculus is not surprising, because F_ω is frequently used as a meta-language for reasoning about formal logic and semantics.

FLINT provides a common compiler infrastructure that can be quickly adapted to generate compilers for new general-purpose or domain-specific languages. With its single unified type system, FLINT serves as a great platform for reasoning about cross-language interoperations. FLINT types act as a glue to connect language features that complicate interoperability, such as mixed data representations, multiple function calling conventions, and different memory management protocols. In addition, because all runtime representations are determined by FLINT types, languages compiled under FLINT can share the same system-wide garbage collector and foreign function call interface (to the low-level C code). Finally, because it has a more expressive type system, FLINT code can also serve as (or translated into) more powerful executable content than Java VM code, making all HOT programs internet ready.

[1]The acronym "HOT" is coined by Bob Harper, and is widely publicized by Phil Wadler in his recent editorial [40] for Journal of Functional Programming.

The FLINT system is being developed at Yale University, using the infrastructure in the type-based version of the SML/NJ compiler [37]. A preliminary implementation of the FLINT intermediate language has been incorporated into the working releases of the SML/NJ compiler since version 109.24 (January 9, 1997). The resulting compiler handles the entire SML'97 [22] plus MacQueen-Tofte higher-order modules [21]. It also gives better performance (about 20% speedup on benchmarks that involve recursive and mutable types) than the older versions of the SML/NJ compiler [37]. New front ends for other languages (e.g., Safe C, Haskell, Java) are under active development.

In the rest of this paper, we first give an introduction to the basic architecture of the FLINT system. We then present the current design of our typed common intermediate language, followed by a summary of the main implementation techniques we used to compile this intermediate language. We further show how different general-purpose or application-oriented languages might be translated into FLINT, and finally, we discuss the related work, and then conclude.

2 The FLINT Architecture

The FLINT system, as shown in Figure 1, is organized around a strongly typed intermediate language also named FLINT. Programs written in various source languages are first fed into a language-specific *front end* which does parsing, elaboration, type-checking, and pattern-match compilation; the source program is then translated into the FLINT intermediate format. The *middle end* does conventional dataflow and loop optimizations [1, 39], local and cross-module type specializations, and λ-calculus-based contractions and reductions [3]; it then produces an optimized version of the FLINT code. The *back end* compiles FLINT into machine code through the usual phases such as representation analysis [34] (to compile polymorphism), safe-for-space closure conversion [36] (to compile higher-order functions), register allocation, instruction scheduling, and machine-code generation [8]. All the compilation stages are deliberately made independent of each other so that they may be pieced together in different ways for different languages.

The runtime system provides support to system-wide garbage collection, foreign-function call interface, and connections to lower-level operating sys-

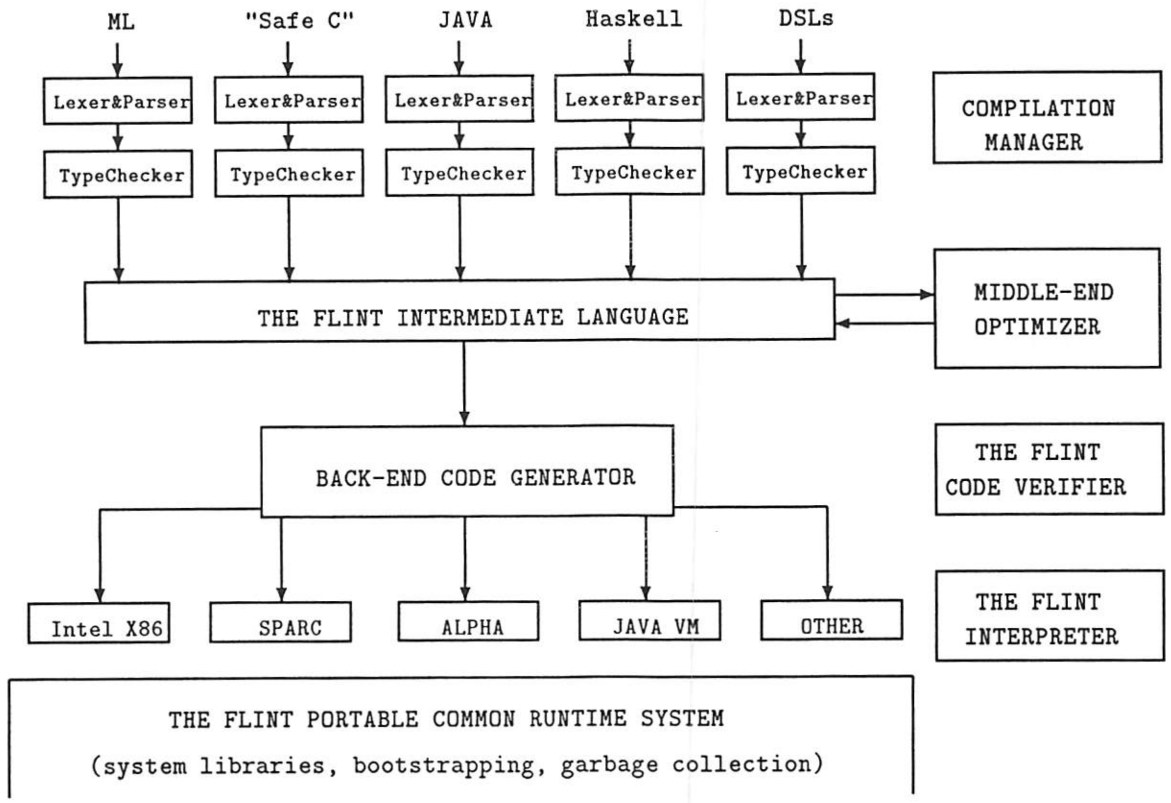


Figure 1: Top-Level Structure of the FLINT System

tem services. Our current implementation borrows SML/NJ's runtime system [30, 2, 18] which runs under all major machine platforms. We plan to extend it to support new services such as dynamic linking and bytecode execution.

It is important to emphasize the modular nature of the overall compiler structure. The FLINT intermediate language nicely separates the language-dependent front end from the language-independent back end. Compiler optimizations done at the middle end are always performed as FLINT-to-FLINT program transformations (so FLINT must be designed as suitable for optimizations). This orgnization also allows FLINT to be used as an advanced executable content language, just like the Java VM bytecode [20]. Here, the front end and the middle end can be thought as a source-to-FLINT compiler; the back end and the runtime system are some kind of *just-in-time* compiler and virtual machine. Because FLINT is designed as a compiler intermediate language, compiling into FLINT does not incur any efficiency loss as the stack-based Java bytecode.

Another important aspect is on the organization of the back end code generator. To keep FLINT's type system simple, we currently let the back end handle the compilation of polymorphism (i.e., representation analysis [19, 34]) and higher-order functions (i.e,. closure conversion [36]). However, there is no reason why we can't propagate and preserve type information throughout the back end. In fact, we intend to propagate the type information into the assembly or machine code to guide sophisticated instruction scheduling and to generate Necula-Lee style proof-carrying code [26].

To construct a compiler for a new application language, we only need to write a new front end that translates the source program into the FLINT intermediate code. If the language is embedded inside another general-purpose language, we simply modify the front end for the host language to support the domain-specific aspects. Most of the time, new primitive operators and type constructors must be added into the intermediate language to support the corresponding surface language constructs; the middle-end optimizer must also be tailored to support the corresponding domain-specific program analysis and transformations. We believe that majority of the domain specific features can be abstracted into a set of algebraic data types, where each consists of a set of primitive types, primitive operators, and proper rewriting rules (i.e., axioms). So even the process of modifying the intermediate language and the middle-end optimizer can be automated.

3 Typed Intermediate Format

Using common intermediate languages to share compiler infrastructure is not a new idea. Many existing compilers, such as GNU GCC, Stanford's SUIF [12], and U. Washington's Vortex [7], all use some kind of shared intermediate format for multiple source languages. In addition, the C programming language has been used as the *de facto* standard target language for a long time. Since all these are mainly designed for conventional imperative languages, none of them directly support higher-order functions or advanced polymorphic type system.

FLINT is designed as a *strongly typed* common intermediate format for HOT languages. There are many advantages in making the intermediate language type-safe. First, a rigorous type system can be used to reason about the safety of a program, even at the intermediate language level. This is particularly important for applications that must be as secure and mobile as the Java VM code. Second, type information makes it possible to reason about principled interoperability among different languages. In fact, because all data representations and function calling conventions are decided based on a uniform type system, it is possible to make programs of different surface languages share the same runtime system (with the same garbage collector and foreign function call interface). Finally, type information have proven invaluable for efficient compilation of statically typed languages such as ML and Haskell [19, 28, 37, 38]; types are also useful for debugging compilers and proving properties of programs.

3.1 Rationale

The current FLINT language is designed based on the following principles:

- *Strong and explicit typing.* ML-like languages often have very tricky type inference problems. Having an explicitly typed intermediate language leaves the type inference issues completely to the front end.

- *Simple and well-defined semantics.* The intermediate language must be simple, clean, and semantically well-founded in order to be used as a common target language.

- *Expressiveness.* In order to support multiple HOT languages, the FLINT type system must be rich enough to express HOT features such as higher-order functions, ML-like polymorphism, and higher-order modules.

- *Pay-as-you-go efficiency.* The intermediate language must, of course, be compiled to generate efficient code. Furthermore, simple, first-order, monomorphic functions should be compiled as efficiently as in C or assembly languages, even though the presence of polymorphic functions might complicate data representations and function calling conventions.

- *Optimization ready.* The compiler middle end performs various kinds of optimizations on the intermediate code. For this reason, the intermediate representation must be compatible with all standard program analysis and transformations [3, 1]. The intermediate language should also contain explicit loop (and recursion) construct to support sophisticated loop optimizations.

- *System-programming friendly.* The intermediate language must provide excellent support to low-level system programming such as safe type-cast, dynamic types, and bit-manipulation primitives. It should also contain a subset of language features that can be used to write real-time programs (e.g., code fragments that do not involve garbage collections).

- *Extensible.* The intermediate language must be easily extended to support other advanced or domain-specific language features (e.g., concurrency, objects, and user-defined datatypes).

To summarize, what we want is a intermediate language that behaves like a strongly typed *assembly* language. It should be high-level enough to express polymorphism and higher-order functions but low-level enough to support all standard optimizations.

3.2 Background

The core language of FLINT is a predicative variant of the Girard-Reynolds polymorphic λ-calculus F_ω [9, 31], with the term language written in the A-normal form [33]. In the following, we first give a introduction about F_ω, and then formally define the Core-FLINT language.

The standard Girard-Reynolds polymorphic calculus F_ω is often defined as follows:

$$\begin{array}{llcl}
(kinds) & \kappa & ::= & \Omega \mid \kappa_1 \to \kappa_2 \\
(types) & \sigma & ::= & t \mid \sigma_1 \to \sigma_2 \mid \forall t :: \kappa.\sigma \\
 & & \mid & \lambda t :: \kappa.\sigma \mid \sigma_1[\sigma_2] \\
(terms) & e & ::= & x \mid \lambda x : \sigma.e \mid @e_1 e_2 \\
 & & \mid & \Lambda t :: \kappa.e \mid e[\sigma]
\end{array}$$

The calculus contains three syntactic classes: kinds (κ), types (σ), and terms (e). Here, kinds classify types, and types classify terms. The extra "kind" hierarchy is used to regulate and define well-formed types. In F_ω, both simple types (e.g., functions, records, integers) and polymorphic types (e.g., $\forall t :: \kappa.\sigma$) have kind Ω; higher-order types (or really, type functions) such as $\lambda t :: \kappa.\sigma$ has kind $\kappa \to \kappa'$, if σ belong to kind κ'. A higher-order type σ_1 can be applied to another type σ_2, written as $\sigma_1[\sigma_2]$.

At the term level, in addition to the usual lambda abstraction and application, F_ω also support explicit type abstraction and type application (written as $\Lambda t :: \kappa.e$ and $e[\sigma]$). Every type abstraction term such as $\Lambda t :: \kappa.e$ has the polymorphic type $\forall t :: \kappa.\sigma$, assuming term e has type σ.

For example, an F_ω function $f = \Lambda t :: \Omega.\lambda x : t.x$ would have type $\sigma_0 = \forall t :: \Omega.t \to t$. In the standard F_ω, the polymorphic type such as σ_0 is still considered to have kind Ω, so expressions such as "$@(f[\sigma_0])f$" would type check, and yield type σ_0.

Because F_ω supports a very general kind of higher-order polymorphism, it is commonly used as the meta-language to reason about formal logic and semantics. In fact, many advanced languages such as ML and Haskell can be embedded into the F_ω-like calculus.

3.3 The Core Language

The core language of FLINT is based on the standard F_ω, but with the following three important changes:

- In standard F_ω, polymorphic types are treated same as monomorphic types, and they both have kind Ω. This complicates the semantics and makes the calculus *impredicative*. Following Harper and Morrisett [15], we split the type hierarchy into two levels: a *constructor* level characterizes the monomorphic types (and type functions), and a *type* level expresses the polymorphic types. "Kind" is now used to classify "constructors" only; polymorphic types such as the previous σ_0 no longer belongs to kind Ω. So expressions such as "$@(f[\sigma_0])f$" will no longer type check in our predicative variant.
- The call-by-value term language is split into two levels as well, with values denoting simple variables or constants. The usual term expressions must now satisfy new syntactic restrictions as standard A-normal forms [33]. More specifically, each function application (or type application) can only refer to values (as $@v_1 v_2$). The standard F_ω function application term $@e_1 e_2$ is rewritten (according to call-by-value semantics) into a nested `let` expressions followed by the actual value application.
- A new product kind $\kappa_1 \otimes \kappa_2$ is added into the kind language to express a sequence of type constructors. The product kind makes it possible to define type functions that takes a sequence of type constructors as argument and returns another sequence as the result. This is useful to express the parameterized modules such as ML higher-order functors [21].

The Core FLINT contains the following five syntactic classes: kinds (κ), constructors (μ), types (σ), terms (e), and values (v):

$$\begin{array}{llcl}
(kinds) & \kappa & ::= & \Omega \mid \kappa_1 \to \kappa_2 \mid \kappa_1 \otimes \kappa_2 \\
(con's) & \mu & ::= & t \mid \texttt{Int} \mid \to(\mu_1, \mu_2) \\
 & & \mid & \lambda t :: \kappa.\mu \mid \mu_1[\mu_2] \\
 & & \mid & \otimes(\mu_1, \mu_2) \mid \Pi_1 \mu \mid \Pi_2 \mu \\
(types) & \sigma & ::= & T(\mu) \mid \forall t :: \kappa.\sigma \mid \sigma_1 \to \sigma_2 \\
(terms) & e & ::= & v \mid \lambda x : \sigma.e \mid @v_1 v_2 \\
 & & \mid & \Lambda t :: \kappa.e \mid v[\mu] \\
 & & \mid & \texttt{let } x = e_1 \texttt{ in } e_2 \\
(values) & v & ::= & x \mid i
\end{array}$$

Here, kinds classify constructors, and types classify terms and values. Constructors of kind Ω now only name monotypes. The monotypes are generated from variables, `Int`, through the constructors $\to$. As in F_ω, the application and abstraction constructors correspond to the function kind $\kappa_1 \to \kappa_2$.

The pairing and selection constructors (i.e., $\otimes$, Π) correspond to the product kind $\kappa_1 \otimes \kappa_2$. Types in Core-FLINT include the monotypes, and are closed under function spaces, and polymorphic quantification. We use $T(\mu)$ to denote the type corresponding to the constructor μ (which must be of kind Ω). As in F_ω, the terms are an explicitly typed λ-calculus (but in A-normal form) with explicit constructor abstraction and application forms. We intentionally included the primitive constructor `Int` and the primitive constant i to show how the core calculus might be extended into a more complete languages.

The static semantics of Core-FLINT, given in Figure 2, consists of a collection of rules for constructor formation, constructor equivalence, type formation, type equivalence, and term formation. The term formation rules are in the form of $\Delta; \Gamma \vdash e : \sigma$ where Δ is a kind environment mapping type variables to kinds, and Γ is the type environment mapping term variables to types. Harper and Morrisett [15, 23] have shown that type checking for predicative F_ω-like calculus is decidable, and furthermore, its typing rules are consistent with the standard call-by-value operational semantics.

3.4 The Full Language

In order to make FLINT as simple as possible, we let the front end deal with many higher-level language constructs. For example, the front end for ML can translate higher-order modules into the Core-FLINT-like calculus [35, 14] in a type-preserving way, thus completely eliminating the need of module constructs from the intermediate language. Similarly, type classes in Haskell can also be embedded into F_ω through explicit dictionary passing.

The complete FLINT language still contains many more type and term constructs than the core languages. Because FLINT is an explicitly typed language, adding new type constructors into FLINT does not involve any type reconstruction problem. In the following, we summarize the main features in our current design:

- A `letrec` construct at the term level to allow the declaration of mutually recursive functions.

- A "sum" type constructor at the constructor level to represent ML-like concrete datatypes. Manipulating values of sum types are done through a set of injection functions plus a "switch"-based projection function.

- A recursive operator at the constructor level to allow definitions of recursive type constructors (e.g., `List`). At the term level, two primitive operators, `roll` and `unroll`, converts values of recursive types into those of the underlying sum types.

- A primitive exception type `Exn` at the constructor level and a pair of term-level constructs: "`raise` v" would raise the exception v, and "`try` e `handle` v" would run the expression e, if any exception is raised, the handler v is called.

- An `Abs` constructor at the constructor level and a pair of primitives `pack` and `unpack` at the term level, with the following kind and type signatures:

  $$\texttt{Abs} :: \Omega \to \Omega$$

  $$\texttt{pack} : \forall t :: \Omega . T(t) \to T(\texttt{Abs}(t))$$
  $$\texttt{unpack} : \forall t :: \Omega . T(\texttt{Abs}(t)) \to T(t)$$

  Every source-level abstract type t is represented in the form of $\texttt{Abs}[\mu]$ inside FLINT, where μ is the internal representation type (hidden from the programmer). The representation types are useful when pickling values of abstract types.

Almost all the rest FLINT constructs can be expressed using the same "signature" form as the above `Abs` primitives. Each signature defines a primitive type constructor at the constructor level and a set of primitive constants and operators at the term level. The primitive functions often satisfy a set of axioms that can be used to optimize the term-level expressions. Our current implementation hardwires the axioms into the middle-end optimizer, but we plan to automate this process in the future.

The FLINT language also includes primitives such as N-bit integers (trapping or non-trapping), N-bit words, N-bit characters (ascii or unicode), N-bit floating-point numbers, strings, boolean types, boxed reference cells, array, packed arrays, vectors, packed vectors, mono arrays and mono vectors, ML-like immutable records (nested or flat), first-class continuations, control continuations (used by CML [29]), suspensions (or thunks, to support lazy evaluations).

In the long term, we plan to add type dynamic and some form of F-bounded quantification to support object-oriented languages such as Java. Type dynamic would also make it possible to translate dynamically typed languages such as Scheme into

Constructor Formation and Constructor Equivalence:

$(v/i/fn)$
$$\frac{}{\Delta \uplus \{t :: \kappa\} \triangleright t :: \kappa} \qquad \frac{}{\Delta \triangleright \mathtt{Int} :: \Omega} \qquad \frac{\Delta \triangleright \mu_1 :: \Omega \quad \Delta \triangleright \mu_2 :: \Omega}{\Delta \triangleright \rightarrow(\mu_1, \mu_2) :: \Omega}$$

$(cfn/capp)$
$$\frac{\Delta \uplus \{t :: \kappa_1\} \triangleright \mu :: \kappa_2}{\Delta \triangleright (\Lambda t :: \kappa_1.\mu) :: \kappa_1 \rightarrow \kappa_2} \qquad \frac{\Delta \triangleright \mu_1 :: \kappa' \rightarrow \kappa \quad \Delta \triangleright \mu_2 :: \kappa'}{\Delta \triangleright \mu_1[\mu_2] :: \kappa}$$

$(cprod)$
$$\frac{\Delta \triangleright \mu_1 :: \kappa_1 \quad \Delta \triangleright \mu_2 :: \kappa_2}{\Delta \triangleright \mu_1 \otimes \mu_2 :: \kappa_1 \rightarrow \kappa_2} \qquad \frac{\Delta \triangleright \mu :: \kappa_1 \otimes \kappa_2}{\Delta \triangleright \Pi_i \mu :: \kappa_i} \quad (i = 1, 2)$$

$(cequiv)$
$$\frac{\Delta \uplus \{t :: \kappa'\} \triangleright \mu_1 :: \kappa \quad \Delta \triangleright \mu_2 :: \kappa'}{\Delta \triangleright (\Lambda t :: \kappa'.\mu_1)[\mu_2] \equiv [\mu_2/t]\mu_1 :: \kappa} \qquad \frac{\Delta \triangleright \mu_1 :: \kappa_1 \quad \Delta \triangleright \mu_2 :: \kappa_2}{\Delta \triangleright \Pi_i(\mu_1 \otimes \mu_2) \equiv \mu_i :: \kappa_i} \quad (i = 1, 2)$$

Type Formation and Type Equivalence:

$(tform)$
$$\frac{\Delta \triangleright \mu :: \Omega}{\Delta \triangleright T(\mu)} \qquad \frac{\Delta \triangleright \sigma_1 \quad \Delta \triangleright \sigma_2}{\Delta \triangleright \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Delta \uplus \{t :: \kappa\} \triangleright \sigma}{\Delta \triangleright \forall t :: \kappa.\sigma}$$

$(tequiv)$
$$\frac{\Delta \triangleright \mu_1 :: \Omega \quad \Delta \triangleright \mu_2 :: \Omega}{\Delta \triangleright T(\rightarrow(\mu_1, \mu_2)) \equiv T(\mu_1) \rightarrow T(\mu_2)}$$

Term Formation:

$(value)$
$$\frac{}{\Delta; \Gamma \vdash i : \mathtt{Int}} \qquad \frac{}{\Delta; \Gamma \vdash x : \Gamma(x)}$$

(fn/app)
$$\frac{\Delta; \Gamma \uplus \{x : \sigma_1\} \vdash e : \sigma_2}{\Delta; \Gamma \vdash \lambda x : \sigma_1.e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Delta; \Gamma \vdash v_1 : \sigma' \rightarrow \sigma \quad \Delta; \Gamma \vdash v_2 : \sigma'}{\Delta; \Gamma \vdash @v_1 v_2 : \sigma}$$

(let)
$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \quad \Delta; \Gamma \uplus \{x : \sigma_1\} \vdash e_2 : \sigma_2}{\Delta; \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \sigma_2}$$

$(tfn/tapp)$
$$\frac{\Delta \uplus \{t : \kappa\}; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \Lambda t :: \kappa.e : \forall t :: \kappa.\sigma} \qquad \frac{\Delta \triangleright \mu :: \kappa \quad \Delta; \Gamma \vdash v : \forall t :: \kappa.\sigma}{\Delta; \Gamma \vdash v[\mu] : [\mu/t]\sigma}$$

Figure 2: The Static Semantics of Core-FLINT

```
type 'a icell = (int * 'a * aux_info) ref      (* internal hash-cell *)

datatype tkindI
  = TK_TYC                                     (* the monotype kind *)
  | TK_SEQ of tkind list                       (* the sequence kind *)
  | TK_FUN of tkind * tkind                    (* the function kind *)
  | .....................

and tycI
  = TC_VAR of DebIndex.index * int             (* tyvar in de Bruijn notation *)
  | TC_PRIM of PrimTyc.primtyc                 (* primitive tycons *)
  | TC_FN of tkind list * tyc                  (* constructor abstraction *)
  | TC_APP of tyc * tyc list                   (* constructor application *)
  | TC_SEQ of tyc list                         (* sequence of tycons *)
  | TC_PROJ of tyc * int                       (* projection on sequence *)
  | TC_FIX of (tkind * tyc) list * int         (* recursive tycon *)
  | TC_ABS of tyc                              (* abstract tycon *)
  | TC_IND of tyc * tycI                       (* tyc memoization node *)
  | TC_ENV of tyc * int * int * tycEnv         (* tyc suspension *)
  | .....................

and ltyI
  = LT_TYC of tyc                              (* monotype *)
  | LT_STR of lty list                         (* structure record type *)
  | LT_FCT of lty * lty                        (* functor arrow type *)
  | LT_POLY of tkind list * lty                (* polymorphic type *)
  | LT_IND of lty * ltyI                       (* lty memoization node *)
  | LT_ENV of lty * int * int * tycEnv         (* lty suspension *)
  | .....................

withtype tkind = tkindI icell                  (* hash-consed tkindI cell *)
     and tyc = tycI icell                      (* hash-consed tycI cell *)
     and lty = ltyI icell                      (* hash-consed ltyI cell *)
     and tycEnv = ......                       (* tyc environment *)
```

Figure 3: Representing Kinds, Constructors, and Types

FLINT. We also intend to extend FLINT to cover more interesting representation types. Since most of these are just new primitive constructors and functions, the overall structure of the FLINT language remains to be simple and small.

3.5 Implementations

One challenge in implementing the FLINT intermediate language is to represent constructors and types compactly and efficiently. Type-based analysis often involve operations such as type application, normalization, and equality test. Naive implementation of these operations would lead to duplicate copying, redundant traversal, and extremely slow compilation.

We use the following techniques to optimize the representations of kinds, constructors, and types (see Figure 3 for a fragment of the FLINT definitions, written as ML datatype definitions). First, we represent all type variables as de Bruijn indices [6]. Under de Bruijn notations, all constructors and types have unique representations.

We then *hash-cons* all the kinds, constructors, and types into three separate hash-tables. Each kind (**tkind**), constructor (**tyc**), or type (**lty**) is represented as an internal hash cell (or *icell*). Each *icell* is a reference cell that contains three pieces of information: an integer hash code, a term, and a set of auxiliary information (**aux_info**). The **aux_info** for constructors and types maintains two attributes: a flag that shows whether it is already in the normal form, and if it is in the normal form, a set of its free type variables. Constructing a new type (or constructor) under this representation would involve: (1) calculating the hash code from its descendants; (2) look up the hash-table, if it is already in, we are done; otherwise, calculate the `aux_info`, and install the new *icell* into the hash-table.

Finally, to make type reduction *lazy*, we use Nadathur's *suspension* notations [24, 25] to represent the intermediate result of *unevaluated* type applications. Intuitively, a type suspension such as LT_ENV(t, i, j, e) is a quadruple consisting of a term t with two indices and an environment. The first index i indicates an embedding level with respect to which variable references have been determined within the term, and the second index j indicates a new embedding level [25]. The environment e determines the bindings for the type variables.

Figure 3 gives parts of the definitions of FLINT kind (`tkind`), constructor (`tyc`), and type (`lty`) using SML datatype definitions. Here, constructor abstraction `TC_FN` and polymorphic type `LT_POLY` all abstract or quantify over a list of type variables; each type variable TC_VAR(i, j) is represented as a de Bruijn index i plus an integer j that indicates the exact position in the corresponding list. Suspension terms are denoted as `TC_ENV` and `LT_ENV`; when a suspension t is reduced, it will be replaced by a memoization node (i.e., `TC_IND` or `LT_IND`). Each memoization node contains a pair: the reduction result t_n and the original term t_o. We keep the original term in the memoization node so that future creations of term t_o can be directly hash-cons-ed to the same memoization node (which requires checking equality against t_o), thus saving unnecessary reductions.

The combination of these techniques have proven to be very effective. With *icell*-based hash-consing and memoization, common operations such as equality test, testing if a type is in the normal form, and finding out the set of free variables, can all be done in constant time. With the use of suspension terms, type application is always done on a *by-need* basis, and once it is done, the result will be memoized for future use. Our preliminary measurements have shown that on heavily functorized applications such as SML/NJ Compilation Manager [4], our optimized implementation is an order-of-magnitude faster (in compilation time) than naive implementations.

Representing type variables as de Bruijn indices does have its drawback. For example, the type-based manipulation becomes much harder to program. A simple beta-reduction such as $v[\mu]$ where $v = \Lambda t :: \kappa.e$ requires adjustment of all type variables occurred free in e; furthermore, if t occurs with some type abstractions, then μ must be adjusted as well.

4 Compiling FLINT

The FLINT code is compiled in two steps. First, the middle end performs a series of conventional control and data flow optimizations. All optimizations are type-preserving so the resulting FLINT code will still type-check under the same typing rules. Because FLINT terms are always in the A-normal form, all CPS-based optimizations [3] apply to FLINT as well. Apart from the presence of polymorphism and higher-order functions, the resulting FLINT code should be very close to the low-level machine languages.

After the optimizations, the back end uses flexible representation analysis [34] to compile polymorphism and safe-for-space closure conversion to compile higher-order functions [36]; it then does the standard register allocation, instruction scheduling, and machine code generation [8].

In the rest of this section, we sample several important techniques used in our compiler back end.

4.1 Type Specialization

Because polymorphic functions are often more expensive than monomorphic functions, the middle end of our compiler performs several rounds of *type specialization* to decrease the degree of polymorphism. The basic idea can be illustrated by the following example:

```
let f = Λt :: Ω.λx :: T(t).x
  in let g = Λs :: Ω.λy :: s.@(f[s])y
      in ... g[Int] ... g[Int] ...
```

Here, assume function f and g are only called as shown, then we can rewrite the above programs into the following:

```
let f' = λx :: T(Int).x
  in let g' = λy :: T(Int).@f'y
      in ... g' ... g' ...
```

Both f and g now become monomorphic functions. This transformation can be carried out through a bottom up traversal: because function g is only applied to `Int`, g can be specialized to `Int` first; after this, f can be specialized in the same way.

4.2 Lambda Reduction

Type specialization will only be most effective if it is combined with conventional dataflow optimizations such as dead code elimination, common subexpression elimination, constant folding, constant propagation, and loop invariants. The middle-end optimizer does all of these. The lambda contraction phase is also a good place to carry out domain specific program analysis and program optimizations.

4.3 Representation Analysis

One novel aspect in our back end is to use the new *flexible representation analysis* technique [34] to compile the polymorphic functions and functors. Under flexible representation analysis, recursive and mutable data objects can use unboxed representations without incurring expensive runtime cost on heavily polymorphic code. In contrast, the *coercion-based* approach used in Gallium [19] and SML/NJ [37] does not support unboxed representations on recursive and mutable objects; the *type-passing* approach used in TIL [38] does handle all data objects, but it involves heavy-weight runtime type analysis and code manipulations.

4.4 Closure Conversion

After the polymorphism is eliminated, we use an efficient and safe-for-space closure conversion algorithm [36] to compile the higher-order functions. Our algorithm exploits the use of compile-time control and data flow information to optimize closure representations. By extensive closure sharing and allocating as many closures in registers as possible, our closure conversion algorithm not only gives good performance but also satisfies the strong *safe for space complexity* rule [3], thus achieving good asymptotic space usage.

5 Translation into FLINT

To demonstrate the power of the FLINT language, we have built a new front end that translates the entire SML'97 [22] plus MacQueen-Tofte higher-order modules [21]) into our typed common intermediate format. This new front end and the FLINT middle end have been incorporated and released as part of the Standard ML of New Jersey compiler since version 109.24 (January 9, 1997). Translation from the Core-ML-like (or Core-Haskell-like) language to FLINT is same as the standard embedding of ML into F_ω [13]; other features such as ML datatypes are translated into FLINT type constructors. Compilation from SML higher-order modules to FLINT is quite a challenge because higher-order modules involve the use of dependent types which, in general, cannot be expressed as F_ω-like polymorphism.

Fortunately, ML-style higher-order modules have a clean phase-distinction property; the module language is completely separate from the core language. In a companion paper [35], we present a type-directed translation of the MacQueen-Tofte higher-order modules into the Core-FLINT like language. The basic idea of our algorithm is like this: we notice that every ML module can be split into a *type* part and a *value* part; the type (value) part of a structure includes all its type (value) components plus the type (value) parts of its structure and functor components; the type part of a functor is an higher-order type function from the type part of its arguments to that of its result; the value part of a functor can be viewed as a polymorphic function quantified over the type part of its arguments; functor applications can thus be expressed as a combination of type application and value application as in the Girard-Reynolds calculus. The detailed algorithm can be found in the companion paper [35].

The fact that ML-style higher-order modules can be embedded into FLINT is a good indication of FLINT's expressive power. We are currently working on translations of other source languages such as Haskell, Java, Safe C. Translating Haskell into FLINT is not much different from translating the Core-SML language. Two distinct features of Haskell are *type class* and *lazy evaluation*. Type class can be eliminated by explicit dictionary-passing, done by the type checker in the front end. Lazy evaluation requires the use of FLINT primitives, *delay* and *force*, to make the evaluation explicit. Translating Java into FLINT is less trivial, but it boils down to what kind of encodings [5] we use to model the Java objects.

We believe that FLINT is a sufficiently rich intermediate language that can be used to handle many interesting application languages. While building a new front end will not be completely trivial, it is definitely much easier than translating into C or building a compiler from scratch. If we consider C as a common intermediate format for conventional imperative languages, FLINT plays the same role but for modern HOT languages.

6 Related Work

Common intermediate format has been an active research area in the past. Many existing compilers such as GNU's GCC, Stanford's SUIF [12], and U. Washington's Vortex [7] all use some kind of shared intermediate representations for multiple source languages. In addition, the C programming language has been used as the *de facto* common intermediate format for a long time. Of course, none of these intermediate languages are strongly typed, and neither do they support advanced HOT languages such as ML.

One example of typed intermediate format is the Java VM bytecode [20]. Like FLINT, the Java bytecode can be statically type-checked, though its type system is not as formalized as the F_ω calculus. Because the Java bytecode is originally designed for Java only, it does not directly support common HOT language features such as higher-order functions and polymorphic functions.

Typed intermediate languages have gotten a lot of attentions in the ML community lately. Several ML compilers, e.g., Gallium [19], SML/NJ [37], and TIL [38], all maintain explicit type information inside their intermediate languages. Our FLINT compiler is the first that handles the entire SML'97 plus MacQueen-Tofte higher-order modules.

Using the predicative polymorphic λ-calculus to model the type-theoretic semantics of Standard ML was pioneered by Harper and Mitchell [13]. Their XML calculus also includes dependent types to characterize ML module constructs. Harper and Morrisett [15, 23] later proposed to use a predicative variant of F_ω (but extended with `typerec`) to compile ML-like polymorphism. Recently, Harper and Stone [16] gave a new type theoretic account for the entire SML'97; the internal language they use still contain a separate module calculus and translucent types. All these work inspired us to look into the possibility of building a typed common intermediate format based on F_ω.

7 Conclusions

We have presented a new framework for constructing high-quality compilers for multiple advanced (HOT) languages. By compiling different general-purpose and application languages into a single typed intermediate format, some of the "Babel" problems associated with application languages can be nicely resolved. For example, the compiler infrastructure we are building can be quickly adapted to generate compilers for new application languages. Also, languages compiled under FLINT can interact with each other based on their static type information. They may also share a single runtime system with system-wide garbage collector and foreign function call interface.

Although the FLINT compiler has been incorporated and released with the SML/NJ compiler for a while, the design of the FLINT language is still at its very early stage. In fact, some important features such as objects and type dynamic are still not supported well. In the future, we plan to gain more experience about application languages, and to expand and evolve FLINT into a more mature intermediate language.

8 Acknowledgments

This research was sponsored in part by the DARPA ITO under the title "Software Evolution using HOT Language Technology," DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

9 Availability

A preliminary implementation of the FLINT intermediate language is now used by (and released with) the Standard ML of New Jersey (SML/NJ) compiler. SML/NJ is a joint work by AT&T, Lucent, Princeton and Yale; both the software and the source code are available via anonymous FTP from :

```
ftp.research.bell-labs.com/pub/smlnj
```

More detailed information and related papers on FLINT can be found at the following WWW page:

```
http://flint.cs.yale.edu
```

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] A. W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, 1990.

[3] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[4] M. Blume. A compilation manager for SML/NJ. as part of SML/NJ User's Guide, 1995.

[5] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Proc. Third Workshop on Foundations of Object Oriented Languages*, July 1996.

[6] N. de Bruijn. A survey of the project AUTOMATH. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Edited by J. P. Seldin and J. R. Hindley, Academic Press, 1980.

[7] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proc. ACM SIGPLAN '96 Conf. on Object-Oriented Programming Systems, Languages, and applications*, pages 83–100, New York, October 1996. ACM Press.

[8] L. George, F. Guillaume, and J. Reppy. A portable and optimizing backend for the SML/NJ compiler. In *Proceedings of the 1994 International Conference on Compiler Construction*, pages 83–97. Springer-Verlag, April 1994.

[9] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur.* PhD thesis, University of Paris VII, 1972.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[11] O. M. Group. The common object request broker: Architecture and specifications (corba). Revision 1.2., Object Management Group (OMG), Framingham,M A, December 1993.

[12] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.

[13] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Prog. Lang. Syst.*, 15(2):211–252, April 1993.

[14] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 341–344, New York, Jan 1990. ACM Press.

[15] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[16] R. Harper and C. Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU-CS-96-136R, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1996.

[17] P. Hudak, S. P. Jones, and P. W. *et al.* Report on the programming language Haskell, a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 21(5), May 1992.

[18] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1996.

[19] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[21] D. MacQueen and M. Tofte. A semantics for higher order functors. In *The 5th European Symposium on Programming*, pages 409–423, Berlin, April 1994. Spinger-Verlag.

[22] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.

[23] G. Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. Tech Report CMU-CS-95-226.

[24] G. Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Duke University, Durham, NC, January 1994.

[25] G. Nadathur and D. S. Wilson. A representation of lambda terms suitable for operations on their intensions. In *1990 ACM Conference on Lisp and Functional Programming*, pages 341–348, New York, June 1990. ACM Press.

[26] G. Necula. Proof-carrying code. In *Twenty-Fourth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1997. ACM Press.

[27] G. Nelson, editor. *Systems programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[28] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *The Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, August 1991. ACM Press.

[29] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 293–305. ACM Press, 1991.

[30] J. H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, January 1993.

[31] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[32] D. Rogerson. *Inside COM: Microsoft's Component Object Model.* Microsoft Press, 1997.

[33] A. Sabry and P. Wadler. A reflection on call-by-value. In *Proc. 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 13–24. ACM Press, June 1996.

[34] Z. Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98. ACM Press, June 1997.

[35] Z. Shao. Typed cross-module compilation. Technical Report YALEU/DCS/RR-1126, Dept. of Computer Science, Yale University, New Haven, CT, July 1997.

[36] Z. Shao and A. W. Appel. Space-efficient closure representations. In *1994 ACM Conference on Lisp and Functional Programming*, pages 150–161, New York, June 1994. ACM Press.

[37] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conf. on Prog. Lang. Design and Implementation*, pages 116–129. ACM Press, 1995.

[38] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 181–192. ACM Press, 1996.

[39] D. R. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Tech Report CMU-CS-97-108.

[40] P. Wadler. Editorial: A HOT opportunity. *Journal of Functional Programming*, 2(7), 1997.

[41] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. Technical Report CS-TR-546-97, Princeton University, Department of Computer Science, Princeton, NJ, April 1997.

Incorporating Application Semantics and Control into Compilation

Dawson R. Engler
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A
engler@lcs.mit.edu

Abstract

Programmers have traditionally been passive users of compilers, rather than active exploiters of their transformational abilities. This paper presents MAGIK, a system that allows programmers to easily and modularly incorporate application-specific extensions into the compilation process.
The MAGIK system gives programmers two significant capabilities. First, it provides mechanisms that implementors can use to incorporate application semantics into compilation, thereby enabling both optimizations and semantic checking impossible by other means. Second, since extensions are invoked during the translation from source to machine code, code transformations (such as software fault isolation [14]) can be performed with full access to the symbol and data flow information available to the compiler proper, allowing them both to exploit source semantics and to have their transformations (automatically) optimized as any other code.

1 Introduction

This paper presents MAGIK, a system that can be used to incorporate both application semantics and control into compilation. MAGIK is motivated by two sets of insights. First, programmer-defined data structures and functions define a semantically rich (albeit syntactically poor) language, built on top of the language the programmer uses to define them. Unfortunately, these ***meta languages*** have not had optimizers: optimization occurs at the lower-level of the programming language, but not at the high-level defined by their interface. Our belief is that since high-level operations are heavy-weight (e.g., they deal with file I/O, window manipulations, transactions, and thread creation), optimizations which understand their semantics offer the hope of significant speed improvements, potentially exceeding the impact of all other compiler optimizations. Second, programming has historically been passive: with the exception of restricted local code transformations provided by macro systems, programmers are limited to writing code, while the power to transform the code has been reserved for compilers. Our belief is that giving programmers safe, ready access to the compilation process will significantly improve the scope of programmer capabilities.

The MAGIK system has been built to test these beliefs. MAGIK provides a simple, modular mechanism for programmers to dynamically incorporate extensions into the MAGIK compiler. User extensions, written in ANSI C, are dynamically linked into MAGIK during compilation. Extensions are given access to MAGIK's intermediate representation (IR) through a set of interfaces that allow them to easily create, delete, and augment IR at compile time. Both this IR and MAGIK are built on top of the lcc compiler [3], which is used to compile the source language (ANSI C). The control MAGIK gives to programmers enables a broad class of optimization and code transformations. This paper presents ten such extensions and sketches of many more.

This paper concentrates on two abilities provided by MAGIK. First, it provides a way for implementors to include domain-specific semantics into compilation. Using this ability, implementors can build both interface optimizers (for speed) and interface checkers (for safety). Interface optimizers exploit application-specific knowledge in order to obtain performance improvements. Such optimizers are applicable to a wide range of interfaces: "bignums", message passing and I/O libraries, math libraries, matrix transformations for graphics, even simple queue operations. From a compiler perspective this ability is useful in any situation where providing a compiler "builtin" would allow more aggressive optimization. From an implementor perspective they are useful in situations where an interface implementor could look at a call or sequence of calls to his implementation and craft specialized call(s) that exploited local uses. For example, a file system implementor can write a optimizer that exploits knowledge of file system operations to perform optimizations such as hiding disk latency by both inserting disk block prefetching commands, and transforming synchronous file I/O operations into asynchronous ones. Interface checkers use application-specific knowledge to enforce stricter semantic checks. For example, by requiring that system call error codes be checked (or inserting such checks) or by ensuring that assertion conditions do not have side effects.

The second main ability MAGIK provides is an easy, modular way to do general code transformations with full access to source information. Using this ability, programmers can instrument code, augment it (e.g., by introducing software fault isolation code [14] or garbage collection reference counters) or enforce invariants about it (e.g., that no pointer casts are allowed). Unlike object code modifiers such as ATOM [11], MAGIK clients are tightly integrated with the source compiler. Performing transformations during the translation from high-level source language to machine code has two important characteristics. First, it provides access to the full semantics of the high-level language, information that source transformers can exploit (or require) during code transformation. Second, the IR produced from these user transformations is a first class citizen, optimized no differently than the IR produced by the compiler itself. As a result, the compiler optimizes these transformations as it would any other code.

This paper is organized as follows: Section 2 discusses related work. Section 3 provides an overview of the system. Section 4 provides two examples of incorporating user-level semantics into optimization, while Section 5 presents two transformations that exploit the information available at source level to augment the code. Section 6 presents further examples of how to use extensible compilation, Section 7 discusses issues in the current system and directions for future work and Section 8 concludes.

2 Related Work

Examples of including application-level information into compilation are compiler-directed prefetching and management of I/O [9] and ParaSoft's Insure++ [8], which can check for Unix system call errors (similar to the MAGIK checker shown in Figure 2). Using a MAGIK-based approach, systems such as these could be built without compiler modifications.

We compare MAGIK to macro systems, semantic-based optimizers, extensible compilers, and object code modifiers.

Macro systems are the most venerable instance of user-level code transformers. An advantage of such systems (Lisp is a good example) over MAGIK is their tight integration with the source language — extensions are typically written in the same language and style as the rest of the application. The main advantage MAGIK provides is power. Macro systems such as Weise and Crew's recent work [16] are restricted to fairly localized code transformations, while MAGIK extensions can perform global transformations across many interface calls, using symbol table and flow graph information provided by the compiler.

Mark Vandevoorde and John Guttag [12, 13] describe a system that provides programmers with a safe way to impart some classes of semantic information to the optimizer. User-level specifications for a restricted functional language are consumed by a theorem prover that optimizes based on the specific situation in which function calls are used. While their system is more automatic than MAGIK, it is less powerful. For instance, MAGIK gives programmers the ability to perform optimizations that appear difficult to express as specifications. The cost of this power is that MAGIK more difficult to use. Further practical experience is needed to determine if MAGIK's added power is worth this cost.

MAGIK follows in the footsteps of the Atom object code modification system [11] (foreshadowed by the object code modifiers of Wall [15] and Srivastava and Wall [10]), which provides users with the ability to modify object code in a clean, simple manner. Atom was one of the first tools to give programmers ready access to the transformational abilities encased in compilers. MAGIK complements this work, and trades the practical generality of dealing with object code for improved information and code efficiency gained by working within a high-level source compiler. Since MAGIK has access to all the information available to the source compiler (e.g., symbol table, flow graph information, high-level semantics) it can derive facts lost at the object code level. For instance, it can easily insert reference counts around all accesses to a particular pointer type; an object code modifier, working solely at the level of loads and stores, cannot. Furthermore, since MAGIK extensions are integrated with the optimization done by the compiler, they can be implemented more efficiently: IR added by an extension is optimized no differently than IR produced from source. In contrast, object code have to both work without much source-level information and cannot bootstrap existing compiler optimizers [15]. An important practical difference between MAGIK and object code modifiers is that MAGIK is significantly easier to implement. The system described in this paper took the author less than a month to implement and it runs on all targets that the base compiler supports (x86, Mips, Sparc). In contrast, duplicating the functionality of ATOM for even a single architecture would require significantly more work (especially on an architecture such as the x86).

There are many compilers designed to support easy addition of optimizations (e.g., SUIF [1]). These system could have been used to implement MAGIK; lcc was chosen because of the author's familiarity with it. To the best of our knowledge, none of these compilers have been used explicitly for extending the optimizer with user-level semantics or transformations.

MAGIK can be viewed as an "Open System" in the spirit of Kiczale's work [7].

Of course, programmers have long performed interface optimizations by hand. The advantages of automated optimization are well known.

3 System Overview

MAGIK provides a framework to extend compilation. User extensions are implemented as dynamically-linked functions. User extensions come in two classes: *code extensions* and *data structure extensions*. Code extensions are invoked at every function definition and are able to enumerate, add, delete, and modify MAGIK's IR as it makes the transition from source language to machine code. Data structure extensions are invoked at every data structure definition and are able to add, delete and modify structure elements. Since compiler internals are in flux, implementation portability is provided by isolating extensions from internal IR details via a set of standardized interfaces; multiple interfaces are provided, specialized to the main domains MAGIK is used in.

A given compilation may use many different extensions. To make the system usable, it is crucial that extension composition is modular. The two main requirements of modularity are that extensions be able to inspect the code produced by others and that extensions can be obliviously composed. MAGIK meets these requirements by providing three different extension types (*transformers*, *optimizers*, and *inspectors*) that correspond to the three main functional uses of extensions. Transformers are used to perform code transformations that do not depend on integration with global optimization (e.g., partially evaluating a C printf call). Optimizers are used to perform iterative optimization and are repeatedly invoked during global optimization until no IR modifications occur. (Optimizers differ from both transformers and inspectors in that they may be invoked multiple times.) Inspectors are similar in functionality to transformers except their placement in the extension pipeline ensures that they see all IR that will be compiled to code.

The main implementation limitation of the MAGIK system is that since lcc provides no global optimization framework optimizers are given only weak data flow information. We are investigating methods of removing this limitation (e.g., by using the SUIF compiler system [1]).

An operational overview of the extension process is as follows:

1. Programmers implement extensions using the MAGIK libraries; these extensions are compiled to object code. The location of this code is either specified to MAGIK using command-line flags or by embedding the location in source files. For instance, header files can specify an extension to optimize the interfaces they define.

2. MAGIK compiles high-level source (ANSI C) to its internal IR in the traditional manner. As MAGIK encounters extension location directives (either as compiler flags or embedded in source) it uses the dld dynamic linker [5] to dynamically link the named extensions into the compiler proper.

3. At every function MAGIK encounters it invokes all code extensions, beginning with transformer extensions. At this point the extensions are free to augment, modify and delete parts of the IR. As part of the global optimization loop, MAGIK calls each optimization extension. These extensions have access to any data flow information computed by the compiler (e.g., use and def sets, values of procedure parameters, etc.) To ensure that code produced by any extension is visible to all others (a requirement for modular composition of different extensions) MAGIK loops through the extensions until no more modifications occur to the IR. A nice result of this organization is that the code produced by an extension is optimized as aggressively as the code produced from application source. After all optimization extensions have run, and no modifications occur, MAGIK runs inspector extensions in their specified order.

Type	C name
V	void
C	signed char
UC	unsigned char
S	signed short
US	unsigned short
I	int
U	unsigned
L	long
UL	unsigned long
F	float
D	double
P	void *

Table 1: MAGIK types (superset of lcc's types).

4. At every structure definition MAGIK invokes all structure extensions. These extensions can add, modify and delete structure entries. Typically these extensions are also paired with code extensions that augment data structure field uses and definitions.

5. MAGIK emits code.

MAGIK's lowest-level IR interface, based closely on that the the underlying compiler (lcc) is terse, simple and portable. Structurally, the IR is a tree language. Leaves are variables, labels, or constants; internal nodes represent operations performed on them (e.g., addition, indirection, jumps, function calls). When operands are created they are associated with a type selected from MAGIK's base types (shown in Table 1). Thereafter, types are implicit: operations infer their own types based on the type of their operands. Any conversions required by ANSI C are performed by MAGIK (e.g., as required by ANSI C a character variable will be converted to an integer before addition with an integer).

User-created IR (type: I_IR) is of a different type than native IR (type: X_IR). This distinction is helpful because user-constructed IR typically requires preprocessing before it can be sensibly incorporated into lcc's internal representation. By exploiting static type-checking, MAGIK can prevent users from blithely intermixing the different representations.

The interfaces are presented in the following tables: routines to allocate, lookup and manipulate symbols in Table 4, routines to construct IR in Table 5, and routines to navigate the IR in Table 2. Higher-level interfaces are discussed in Section 4 and Section 5.

We expect MAGIK to evolve with further experience. To aid iterative design, the current implementation has emphasized simplicity at all levels. MAGIK is built on top of the lcc retargetable ANSI C compiler [3], and uses its IR language as its fundamental interface [4] (higher-level interfaces are crafted on top of this). The regularity and small size of lcc's IR has been a major asset. Importantly, since mapping other IR's to the MAGIK IR and back should be straightforward, it can be realistically used as a basis for defining a standardized, compiler-independent, extension interface similar in availability to ANSI C's standardized libraries.

While the implementation exploits lcc's infrastructure, there is no fundamental tie to lcc. As experience with the system and its uses grows, reimplementations will occur in more aggressive compilers (or, alternatively, MAGIK will be used to enhance the optimization framework of lcc).

One of the common uses of MAGIK is to incorporate new functions as "built-ins" into the compiler. Since there can be tens or (at aggressive sites) hundreds of builtins, it is critical that the extension process itself is efficient. To achieve the required efficiency, MAGIK dynamically links extensions rather than isolating them in sub-processes that communicate via shared memory. In most cases this process has no significant impact on compilation speed. For implementations that wish to remove all overhead (at some cost in reduced flexibility) MAGIK provides an interface that can be used to statically link extensions into the compiler proper (similar to the process of adding device drivers to most operating systems).

The following two sections discuss the interfaces MAGIK provides for incorporating application semantics (e.g., interface optimization and checking) and for general code transformation.

4 Incorporating Application Semantics

As discussed previously, user-level data structures and functions define a high-level language, the semantics of which is unavailable to traditional compilers. MAGIK provides mechanisms that allow applications to construct extensions that can exploit these languages' semantics for improved semantic checking, optimization, and general transformations. The two main constructs of interest are functions and data structures. In the case of functions, clients are mainly interested in two pieces of data-flow information: the location of calls in relation to each other, and the definitions and uses of each call's operands and results. Clients also require semantic information about each call site's operands: their type, whether they are constants, and if so, what their values are. In the case of data structures clients are primarily interested in definitions and uses of structures and their fields.

To make IR manipulations easier, MAGIK exploits the limited information needed in this area to provide a default interface that is simpler than the general MAGIK IR. It includes basic block structures, function calls, details about function arguments and results (e.g., whether they are constants, their type, possible values, etc.) and information about structure accesses. A library of routines are provided that allow clients to add, modify, delete and augment function calls and code easily. Additional routines are provided to search for particular functions and lists of functions in the IR (easing IR navigation), traverse argument lists, and routines that compute the set of variables defined and used by a given call site. Table 6 presents MAGIK's interface for finding, manipulating, and constructing call sites. Table 7 presents MAGIK's interface for finding IR tree patterns, and both structure and structure field uses.

Clients that need access to the full power of MAGIK's IR can, of course, use it; the layering provided by default is intended as syntactic sugar rather than a barrier.

The following subsections present MAGIK's semantic interface and four example clients. The first client exploits MAGIK to perform the general transformation of adding a compiler "builtin" output function that is implicitly aware of its operand types (eliminating the need for printf-style format strings). The second client adds more rigorous semantic checking of Unix system calls by inserting checks around call sites that ignore a system call's return value. The third client ensures that signal handlers call only reentrant functions. Finally, the four extension optimizes RPC call sites by using partial evaluation to generate specialized argument marshaling code.

4.1 Example: adding type-aware functions

ANSI C suffers from the lack of a graceful mechanism to handle poly-typed functions. Programmers are typically reduced to specifying argument types using a manually-constructed type string. This methodology is clumsy and error prone. One of the more painful effects of this lack is that C is one of the few languages in use that does not have type-aware I/O routines.

Figure 1 presents a MAGIK extension that adds a type-aware output routine, output. It works by rewriting all calls to the poly-typed function it defines (output) to call printf using a type string (typestring) it constructs from the type of output's arguments. An operational view is as follows:

1. The extension iterates over all calls to output using the MAGIK functions FirstCall and NextCall.

2. For each callsite, it builds up a printf-style type string by iterating over output's argument list (using the MAGIK functions FirstArg and NextArg) and appending the type of each argument to typestring.

3. After typestring has been constructed, the extension uses RewriteCall to modify the call site to call printf instead of output and inserts typestring as the first argument.

A sample usage:

```
void example(int i, int j) {
  output("i = ", i, "j = ", j);
}
```

While some languages (such as C++) support this capability for simple scalers, our extension can be easily modified to print the fields in aggregate types, freeing programmers from having to tediously write data structure-specific output routines (this functionality was elided for brevity). Extensible code synthesis is powerful. Example uses include the automatic generation of routines to translate data structures between "in-core" and on-disk representations and the construction of linked-list, hashtables, and associative arrays specialized to particular data structure types. A similar technique is used in Subsection 4.4 to construct an efficient argument marshaling routine for a remote procedure call system.

4.2 Example: safe system calls

C and Unix are notorious for using integer error codes to indicate exceptional conditions. C and Unix programmers are notorious for not checking these codes. This problem is a significant one, especially with the prevalence of network computing (where file I/O operations have to be retried with some frequency). Figure 2 presents an an extension that inserts error condition checks around unchecked Unix system calls and prints out errors that occur.

The extension works as follows:

```
/* Add a type-aware output function. */
int RewriteOutput(X_IR c) {
# define MAXARGS 64
    X_IR a;

  /* Foreach callsite, rewrite the output call. */
  for(c = FirstCall(c, "output");
      c != NULL;
      c = NextCall(c, "output")) {

    /* String to hold derived typestring. */
    char typestring[MAXARGS*2+1] = {0};

    /* Foreach argument, create a typestring. */
    for(a = FirstArg(c); a != NULL;
                a = NextArg(a)) {
      switch(OpType(a)) {
      case I: strcat(typestring, "%d "); break;
      case P:
        /* Print strings differently
             than pointers. */
        if(RawPtrType(NodeType(a)) == C)
          strcat(typestring, "%s ");
        else
          strcat(typestring, "0x%p ");
        break;
      /* ... */
      default: panic("Bogus type");
      }
    }
    /* Add newline */
    strcat(typestring, "\n");
    /* Change call to output to call to printf. */
    RewriteCall(c, "printf");
    /* Add typestring as first argument. */
    PushArg(c, Cnststr(typestring));
  }
  return MAGIK_OK;
}
```

Figure 1: Routine to add a type-aware output routine to C

```
/* Add checks to unchecked system calls. */
int RewriteUnix(X_IR c) {
  /* list of all calls we insert checks for */
  char *unixcalls[] = {"read","write","seek",
                          /* ... */0};
  I_IR res, err, stmt;
  char *n;

  /* foreach callsite, rewrite the output call */
  for(res = NULL, c = FirstCallV(c, unixcalls);
     c != NULL; c = NextCallV(c, unixcalls)) {

    n = CallName(c);

    /* If result used, assume it is checked. */
    if(Uses(c))
      continue;
    else
      warn("unchecked system call <%s>\n",n);

    /* Create temp to hold returned value */
    if(!res)
      res = Temp(inttype, MAGIK_REG);

    /* Create IR to assign the return value
      to res. */
    stmt = AddStmt(c,
            Asgn(res, ImportExprRef(c)));

    /* Create a call to error routine; expects
      syscall's name and return code. */
    err = Call("error", voidtype, Cnststr(n),
             res, NULL);
    /* Insert check for syscall failure. */
    AddStmt(stmt,
              IfStmt(Lt(res, Cnsti(0)), err));
  }
  return MAGIK_OK;
}
```

Figure 2: Extension that places error checks around unchecked system calls.

1. It iterates over all calls to the functions listed in the array unixcalls using the MAGIK functions FirstCallV and NextCallV.

2. For each call site it checks if the result of the call is used (using the MAGIK routine Uses). Unfortunately, a use does not guarantee that the call's result is checked — for simplicity, we elide more aggressive checking.

3. For call sites that do not use the result of the system call, the extension creates IR to check the system call's return value and, if it is an error, call an error procedure (error) to print it out. It then inserts this IR into the original IR using AddStmt.

4.3 Example: safe signal handlers

Unix signal handlers represent primitive threads of controls. Unfortunately, they are used by many programmers who are unfamiliar with the dangers of threaded programs. A common mistake made is to call non-reentrant library functions from these handlers. If the application was suspended in the middle of a call to the same function (or to a function that manipulates state it depends on) the application program will, non-deterministically, exhibit incorrect behavior.
To help prevent this class of problems we have defined an extension that prevents calls to non-reentrant functions in a signal handler (the extension's code is elided for brevity). The extension works as follows:

- To trigger checking all signal handlers adhere to the naming convention of prefixing their name with "sig_" (e.g., sig_protection_fault).
- The extension scans for all functions beginning with this prefix and, for each callsite, checks that the call is either to one of a list of known reentrant functions or to a function that is prefixed with sig_. Any call that does not satisfy these requirements is flagged.
- To ensure that only checked handlers are installed as signal handlers it also looks for handler installation calls and checks that they only install functions beginning with the sig_ prefix.

4.4 Example: RPC specialization

Remote procedure call (RPC) is a widely used abstraction in distributed programming. A significant overhead of a general-purpose RPC call is the cost of copying the call's arguments into a message buffer ("argument marshaling"). Figure 3 presents a MAGIK extension that uses partial evaluation to remove the main contributor to this overhead, the interpretation of argument types, by crafting marshaling code specialized to a particular callsite.
The extensions infrastructure is similar to that used to implement output: it scans for calls to rpc and examines the its argument which, syntactically, is a call to a remote function. It decomposes this call into its constituent pieces and then builds marshaling code to copy each argument in the RPC call into a memory vector. It then rewrites the call to rpc to take a pointer to a local copy of the remote procedure along with a pointer to the constructed message buffer and its size. A sample usage is as follows:

```
int k,j,i;
double d;
/* ... */

/* call remote procedure remote_foo */
rpc(remote_foo(j, i, k, d));
```

Of course, this usage can be made prettier by communicating the names of remote procedures to the extension, thereby eliminating the need for the rpc annotation. For simplicity we do not perform this syntactic cleanup (we also ignore result passing).

5 Code transformations

Code transformations involve rewriting or augmenting general code (i.e., unlike the extensions described in the previous section, their domain is not limited to a specific interface). Example code transformations are software fault isolation, the translation of pointers from one representation to another, or the insertion of checks to ensure a pointer use is not nil.
In MAGIK, code transformations are typically implemented by searching for specific IR trees and (possibly) replacing or augmenting them. To make this style of usage easy, MAGIK provides an interface specialized to this domain. IR navigation can be implemented using MAGIK-provided pattern matching routines that iterate over IR, returning all locations that extension-specified IR trees occur at. Rewriting support includes procedures that insert, delete and augment IR subtrees. These routines isolate the programmer from implementation-specific details of

```
/* Find RPC calls and build marshalling code. */
int MarshalGen(X_IR r) {
  /* foreach callsite, rewrite output call */
  for(r = FirstCall(r, "rpc");
      r != NULL; r = NextCall(r, "rpc")) {
    I_IR index, marshalv;
    int offset, sz;
    X_IR a, c;

    /* Allocate marshaling array on stack. */
    marshalv = Array(doubletype, Nargs(c));
    offset = 0;
    /* Remote call is rpc's first argument. */
    c = FirstArg(r);

    /* Store arguments in marshalling vec. */
    for(a = FirstArg(c); a != NULL;
                          a = NextArg(a)) {
      /* ensure correct alignment. */
      offset = roundup(offset, NodeAlign(a));
      sz = NodeSize(a);

      /* Form expression "*(type *)(marshal +
         offset)" where type is typeof(a). */
      index = Index(Cast(Copy(marshalv),
                              Ptr(NodeType(a))),
                Cnsti(offset/sz));

      /* marshalv[offset] = a */
      PushStmt(c,
            Asgn(index, ImportExprCopy(a)));
      /* Add size of argument. */
      offset += NodeSize(a);
    }
    /* Replace rpc call with message send; send
       takes a pointer to a local copy of the
       remote function and the marshal vector
       and size as arguments. */
    c = ReplaceExpr(c,
        Call("send", inttype,
            CallName(c), Copy(marshalv),
                Cnsti(offset), NULL));
  }
  return MAGIK_OK;
}
```

Figure 3: Extension that creates specialized marshaling code based on remote procedure call argument types.

```
/* Used by qsort to compare element sizes. */
static int pack_cmp(void *p, void *q) {
  return FieldSize(*(Field *)p) -
           FieldSize(*(Field *)q);
}

/* Look for structures with "pack_" prefix and
   minimize their storage size by sorting their
   elements by size. */
void Packer(Symbol p) {
  unsigned n;
  Field *fl;

  if(strncmp(StructName(p), "pack_", 5) != 0)
    return;
  /* Get fields */
  fl = ImportFields(p, &n);
  /* Sort them. */
  qsort(fl, n, sizeof fl[0], pack_cmp);
  /* Write them out. */
  ExportFields(p, fl, n);
}
```

Figure 4: Routine to minimize structure size by sorting elements by alignment requirements.

IR modification (e.g., the need to update all pointers to a node that has been used as a CSE).

5.1 Example: structure packing

Dense structure layout can be used to improve locality. Figure 4 presents a data structure extension that rearranges structure fields to reduce structure size. Using the same capabilities extensions can perform many useful structure transformations: fields can be automatically arranged to be endian-neutral and on machines that lack sub-word operations, shorts and chars can be promoted to ints.

6 Extensible compilation: patterns of use

This section delineates some broad classes of extensible compiler uses. Both simple and ambitious examples are included to give a flavor of the range of operations that can be performed. Many of the examples provide programmers with capabilities not previously available.

User semantic optimization As described in Section 4, the languages defined by interface's functions and data structures have not had optimizers that understood their semantics. Since the operations defined by these languages are heavy-weight, providing a mechanism to incorporate this information offers the potential of speed improvements exceeding the impact of all other compiler optimizations.

An example of this style of use is an extension that understands remote procedure call (RPC). When it encounters a series of RPCs, it can aggregate them into a single message (improving throughput) and by looking for definitions of their operands and uses of their results, replace synchronous RPC with asynchronous, and push the call higher in the program text, and the check for completeness right before any use (improving latency). Similar optimizations can be done for file I/O.

Another example is an extension that optimizes calls to a graphics library. Consider a sequence of calls that manipulate a matrix. Using a library-specific extension, it is possible to optimize across these calls, reusing intermediate results they compute, eliminating intermediate copies, and performing cache optimizations across them.

Finally, a "big num" package can optimize across calls to its operations.

Operationally, this approach can provide a performance gain for any situation where a system's implementor could look at a section of code and implement a specialized operation to capture the same functionality. The challenge with the extensions is to codify this knowledge.

Extension of compiler builtins Incorporating knowledge of functions into compilers in the form of "builtins" is profitable both in terms of syntactic sugar and in performance. Unfortunately, the inclusion of builtins requires the intervention (and interest) of compiler writers rather than system implementors. Consequently, it has been put to limited use despite its utility. Using MAGIK, implementors can easily add builtin procedures.

There are many simple routines (sorting, searching, tree and list manipulations) that are constantly reimplemented in order to work on different types. Using MAGIK these routines can be defined once, by an extension, and then used by all application writers. To illustrate this capability we have implemented a simple extension that defines a max procedure that works on any scaler argument type.

To show how MAGIK can be used to define builtin procedures for improved performance we have also written an extension that recognizes the ANSI C memcpy ("memory copy") function. The extension exploits information MAGIK provides to specialize to the local characteristics of each callsite. For example, in the general case, memcpy must treat its operands as unaligned. However, using the semantic information MAGIK provides, the extension can determine when a call site's pointer operands are aligned and specialize accordingly. Additionally, it unrolls and inlines the memory copying loop when the number of bytes to copy is a constant, Static specialization removes runtime selection overhead, and shrinks the function's memcpy footprint (due to the fact that the gaps introduced by non-taken cases is eliminated). These optimizations are profitable in the context of operating system device driver and networking code, which can extensively access fixed-sized quantities of partially unaligned memory.

Partial evaluation A more general form of builtin specialization is full partial evaluation. Using an extensible compiler, both automated systems and programmers construct partial evaluators for important routines. For example, Section 4 described an extension that generated specialized code for RPC marshaling.

Structure awareness The ability to automatically traverse, rearrange, redefine, and augment data structure members enables interesting operations. Data structure traversal allows the definition of structure independent routines for sorting, searching, marshaling, and printing. Control of data structure layout can improve performance by allowing extensions to group member fields that are used close together into the same cache line, improving cache behavior. It can also enhance usability by enabling extensions to abstract away such details as endianness by automatically rearranging structures to be endian neutral. Data structure redefinition can improve speed on machines that do not provide sub-word memory instructions by allowing an extension to replace sub-word sized structure elements with word-sized ones. Data structure augmentation allows functionality enhancements such as automatic addition of bookkeeping fields needed by reference counting garbage collectors.

Added safety MAGIK offers improved software quality in addition to higher performance. Using it, implementors can construct checkers more stringent than provided by the compiler proper as well as inserting code to check for errors at runtime. For example, to ensure stronger pointer safety, MAGIK

```
/* Look for function calls or assignments */
static int HasSideEffect(X_IR c) {
  if(!c)
    return 0;
  else if(Op(c) == ASGN || Op(c) == CALL)
    return 1;
  else
    return HasSideEffect(Left(c))
        || HasSideEffect(Right(c));
}

/* Check that assertions do not contain
     side-effecting optations. */
int AssertCk(X_IR c) {
  for(c = FirstCall(c, "assert");
      c != NULL;
      c = NextCall(c, "assert")) {

      /* The assertion expression is the call's
           first argument. */
     if(HasSideEffect(FirstArg(c)))
        warning("assert has a side-effect\n");
  }
  return MAGIK_OK;
}
```

Figure 5: Routine to guarantee that assertions are free of side-effects.

can be used to construct a code inspector that statically checks the IR generated at compile time to disallow all casts, implicit conversions, and adds runtime checks to guard against over and underflow of numbers, nil and bogus pointers, and out-of-bound array accesses. Figure 5 presents an extension that guards against side-effects in assertion macros.

The ability to insert integrity checks without requiring source modification is a powerful prophylactic measure to guard against errors, and can serve to elevate C (somewhat) to the realm of modern languages.

Passing compiler information to applications Compilers compute much useful information. MAGIK provides an infrastructure that can be used to pass this information to applications. Two example extensions we have built in this spirit are an extension that, given a pointer to a type, returns the alignment of that type (this is useful for memory allocators) and an extension that takes a single argument and indicates whether it is a constant expression (useful in making inline decisions).

Code transformations The ability to augment code is powerful. Using MAGIK's interfaces, applications can implement a vast set of code transformations such as the insertion of reference counting, software address translation (as described in Section 5), or providing protection via software fault isolation [14].

An interesting optimization is to encode the expected result of interface calls in an extension. These "annotations" allow the extension to rearrange code so that the conditional bodies of unexpected cases are moved off of the commonly executed path, thereby improving both instruction prefetching queue and instruction cache utilization.

Exploiting type information The ability to access symbol table information enables operations not typically supported in Algol languages. For example, programmers can use MAGIK to pass types as arguments to functions such as `malloc` so it can track the pointer type it is allocating and be accurate (rather than conservative) in the alignment it provides.

Investigation Ready access to a semantically-rich intermediate language can be used to answer many questions about source-level code. For example, it can be used to verify hypothesis about software engineering by correlating bug reports to how many times an abstraction layer is broken (perhaps by tracking structure accesses) or by correlating ease of modification to the number of intermodule dependencies a source file has. Checks can be inserted to check for the aliasing of pointers to determine what optimizations would be profitable. It can also be used to support graphical performance monitoring in the spirit of Jeffery and Griswold [6] by automatically inserting display calls around interface uses.

Other Uses There are many other uses for extensible compilation. For example, many uses of Atom can also be done using MAGIK(the tradeoff is less generality for more information and optimization). It provides an easy way to incorporate annotations into the optimization phase by looking for annotations in the form of function invocations. It can be used to restrict allowable operations in the input language in order to make it more amenable to optimization. Or it can be used augment the base language with abilities such as exception handling.

With a sufficiently rich intermediate language MAGIK's extension framework can be used to make its compiler into a truly open system, where a variety of implementors can augment its core optimization abilities with new optimizations. In this manner, compiler optimizations would become an order of magnitude easier to disseminate.

7 Discussion

MAGIK attempts to literally make "library design language design." It does this by attacking the three crucial differences between writing a function-level interface and defining an input language and compiler. The first difference is obvious: languages have syntactic sugar, libraries do not. By enabling interface designers to include context- and semantic-sensitive code transformers, sugar can be judiciously added to function interfaces (e.g., as done in the output and rpc examples in Section 4). The second difference is more subtle: languages allow semantic checks that can be difficult for a library to replicate in terms of its implementation language. By giving extensions access to both the symbol table and function-level IR this barrier can be eliminated. Finally, languages can be optimized. Encoding their semantics in a compiler allows a ready implementation of both local (e.g., peephole optimization) and global (e.g., CSE) optimizations. Current compilers are blind to interface semantics, precluding analogous optimizations. MAGIK provides mechanisms that can be used to build interface optimizers that optimize interface primitives as aggressively as source language constructs.

7.1 Interface issues

An interesting research question is determining the design rules for building interfaces that are amenable to language-like optimization techniques. Two principles seem relatively safe. First, high-level optimization is aided by the use of declarative, high-level interfaces that can then be "strength-reduced" to the characteristics of local usage. Second, optimization across interface calls is eased if the result of one interface call is immediately used by another: function call nesting is an ideal way of eliminating data-flow ambiguities. Thoroughly codifying practical precepts will be challenging.
Careful (but, unfortunately, iterative) design of the MAGIK system has allowed us build it so that it is integrated with the infrastructure lcc uses to construct its internal IR. An important result of this integration is that we have been able to use the frontend routines lcc provides for constructing abstract syntax trees. Using this code has two significant benefits. First, it allows users to only specify types when defining constants and symbols: the remaining IR-construction routines can derive required types from context (e.g., Add can determine it is an integer addition by examining its operands). Eliminating the need to explicitly encode types has dramatically simplified MAGIK's code construction interface. Second, lcc's routines are designed to perform implicit conversions as required by the rules for ANSI C. As a result, they type-check their arguments (providing users with safety) and perform coercions as necessary (providing users with convenience).
There are a few challenges to using the current IR system. The first is dealing with IR tree layouts across compiler versions. Layout of IR trees is a fairly volatile implementation feature. Currently, MAGIK decrees an IR interface and layout. The cost of this solution is that future implementations may require extra mapping code to compile their IR to the standardized MAGIK IR and back. An alternative solution is to specify code using a higher-level representation. The main technical challenge of using IR to specify patterns is that functionally identical language expressions may be compiled to structurally different trees. Fortunately, the lcc IR is spare enough that this problem is not difficult: the number of possibilities tends overwhelmingly to one and, in rare cases, two. In fact, the use of a low-level IR can have a significant benefit over both source-level and machine-code matching in this respect since both, in practice, can contain significant numbers of synonyms (e.g., consider the possible ways to get values to and from memory on the x86, or the different but equivalent methods to reference an array element in C). However, while the IR representation has been sufficient for all examples we've wanted to implement, there are times when a less strenuous mechanism of code specification is preferable. We are currently investigating alternatives.

7.2 System limitations

There are a number of limitations with the current system; most were deliberately chosen in order to allow it to be built quickly so that real programmers could use it in the near future, thereby allowing the wheel of iterative design to begin turning with the least amount of delay. Four main limitations are discussed.
First, constructing large pieces of code is tedious.

This would naturally be remedied with language support. A promising avenue is to use the 'C language [2] (designed to construct code dynamically) as a sugary method of dynamically constructing MAGIK IR. 'C solves most of the semantic issues dealing with variable binding, and code construction, leaving us with the fairly straightforward task of modifying it to dynamically emit MAGIK IR rather than executable code.

Second, the current code specification MAGIK interface — the low-level IR of lcc — while simple, is perhaps not the most natural for mainstream programmers. There are tradeoffs in this representation: a low-level IR can be more precise, however, it can also be more complex than necessary. We are investigating the representation of code templates used for matching via language support: here to a modification of the 'C language seems promising.

Third, the system is manual, even for tasks that could be done automatically (e.g., in the spirit of Vandevoorde and Guttag [13, 12]). As we determine which of these tasks are important and common, automation will be added.

Finally, lcc, while simple and easy to modify, is a poor optimizer. We are examining ways to improve its code quality.

7.3 A simple language extension

Exploitation of application semantics is helped if semantics can be clearly and unambiguously indicated. For example, translating shared memory accesses is eased if every such access can be explicitly labeled as "shared." The clean, clear conveyance of semantic information to extensions is a general problem. Fortunately, it has a simple solution: the addition of a new syntax operation to ANSI C (annotation) that is used to create new, scoped type qualifiers. These qualifiers would be syntactically parsed and internally stored in the symbol table but otherwise ignored by the compiler proper — their semantics provided solely by extensions. An example usage:

```
/* add "shared" as a new type qualifier */
annotation shared;
/* Allocate an integer with new type qualifier. */
shared int x;
```

8 Conclusion

This paper has addressed two problems programmers have historically faced. First, the languages they define via interfaces have not been treated as first-class languages. As a result, these languages have had no language-specific semantic checkers, transformers, or optimizers. Second, their programs are passively consumed with little support for active transformation (such as rewriting of structure fields and the addition of profiling code).

The MAGIK system is a first step towards solving these problems. MAGIK provides a modular interface implementors can use to extend compilation. The main interaction is through a set of interfaces that give extensions access to the IR produced from source. MAGIK thus provides a method that system implementors use both to incorporate domain-specific semantics into compilation (thereby enjoying the obvious advantages of automated optimization and checking) and to perform general transformations on the IR produced from source (thereby having both access to high-level semantic information and the resulting transformation code optimized as aggressively as code produced from source).

This paper has presented many example clients of the MAGIK system. Many of these extensions provide capabilities that programmers did not previously have. Future research will involve both extending these capabilities and exploring their consequences.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[2] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, FL, 1995.

[3] C. W. Fraser and D. R. Hanson. *A retargetable C compiler: design and implementation.* Benjamin/Cummings Publishing Co., Redwood City, CA, 1995.

[4] C.W. Fraser and D.R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

[5] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software—Practice and Experience*, 24(4):375–390, April 1991.

[6] Clinton L. Jeffery and Ralph E. Griswold. A framework for execution monitoring in Icon.

Software—Practice and Experience, 24(11):1025–1049, November 1994.

[7] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[8] Adam Kolawa and Arthur Hicken. Insure++: A tool to support total quality software. `http://www.parasoft.com/insure/papers/tech.htm`.

[9] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[10] A. Srivastava and D.W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, March 1992.

[11] Amitabh Srivastava and Alan Eustace. Atom - a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[12] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance.* PhD thesis, M.I.T., 1994.

[13] Mark T. Vandevoorde and John V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *Proceedings of the 1994 ACM/SIGSOFT Foundations of Software Engineering Conference*, 1994.

[14] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 1993.

[15] D.W. Wall. Systems for late code modification. *CODE 91 Workshop on Code Generation*, 1991.

[16] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of PLDI '93*, pages 156–165, Albuquerque, NM, June 1993.

Operation	**Description**
X_IR LeftChild(X_IR n)	Returns n's left child or nil on error.
X_IR RightChild(X_IR n)	Returns n's right child or nil on error.
int OpType(X_IR n)	Returns opcode of n.
int Type(X_IR n)	Returns type of n.
int Align(X_IR n)	Returns alignment of n.
int Size(X_IR n)	Returns size of n.

Table 2: Base IR Interface.

Class	**Examples**	**Prototype**
Arithmetic binary operations.	ADD SUB MUL DIV XOR AND OR	I_IR op(I_IR a, I_IR b)
Arithmetic unary operations.	NEG COM	I_IR op(I_IR a)
Conversions ("convert to *type*").	CVTI CVTD CVTUS	I_IR *op*(I_IR a)
Memory operations.	ADDR INDIR	I_IR *op*(I_IR a)

Table 3: Partial IR-construction Interface. Functions determine the type of opcode to use based on operand type. Conversion conventions are those of ANSI C.

Operation	**Description**
I_IR Local(Type ty)	Creates a local variable of type t and returns its symbol.
I_IR LocalArray(Type ty, int n)	Creates a local array of type t and size n and returns its symbol.
I_IR Global(Type ty)	Creates a global variable of type t and returns its symbol.
I_IR GlobalArray(Type t, int n)	Creates a global array of type t and size n and returns its symbol.
I_IR Cast(I_IR var, Type t)	Creates a copy of symbol var changing its type to t.
I_IR Lookup(char *name)	Lookup symbol for variable name.

Table 4: Symbol construction and manipulation routines (routines to construct new aggregate types are elided).

Operation	**Description**
X_IR Copy(X_IR n)	Create a copy of node n. This function is typically used when adding a new subtree between a node and its child.
I_IR ImportExprRef(X_IR expr)	Import a reference to expr. This reference can then be used as an argument to functions that require a I_IR type.
I_IR ImportExprCopy(X_IR expr)	Import a copy of expr. This copy can then be used as an argument to functions that require a I_IR type.
X_IR AddStmt(X_IR a, I_IR stmt)	Add stmt after node a. Returns stmt.
X_IR PushStmt(X_IR a, I_IR stmt)	Add stmt before node a. Returns stmt.
X_IR DeleteStmt(X_IR stmt)	Remove stmt, returns its successor.
X_IR DeleteExpr(X_IR expr, I_IR replacement)	Delete node expr; replaces the tree with replacement. If replacement is nil, MAGIK will coalesce the tree expr was part of until it is well-formed.
X_IR AddExpr(X_IR a, I_IR b)	Insert b on top of a.
I_IR If(I_IR bool, I_IR stmt)	If bool is true, execute stmt.
I_IR IfElse(I_IR bool, I_IR stmt1, I_IR stmt2)	If bool is true, execute stmt1 otherwise execute stmt2.
I_IR While(I_IR bool, I_IR stmt)	While bool is true, execute stmt.

Table 5: Partial High-level IR construction Interface

Operation	**Description**
X_IR FirstCall(char *name)	Returns pointer to first call of name or nil if none is found.
X_IR FirstCallV(char **namelist)	Returns pointer to first call of any function in namelist or nil if none is found.
X_IR NextCall(X_IR c, char *name)	Returns pointer to next call of name or nil if none is found.
X_IR NextCallV(X_IR c, char **namelist)	Returns pointer to next call of any function in namelist or nil if none is found.
X_IR RewriteCall(X_IR call, char *newname)	Replace name of call to be newname.
X_IR FirstArg(X_IR call)	Return first argument (if any) of call.
X_IR NextArg(X_IR arg)	Get next argument (if any) after arg.
X_IR Arg(X_IR call, int n)	Returns the nth argument of call; returns nil on error.
void PushArg(X_IR call, I_IR arg)	Adds arg as the first argument to call.
void AppendArg(X_IR call, I_IR arg)	Adds arg as the last argument to call.
int NArgs(X_IR call)	Return number of arguments to call.
X_IR ReplaceArg(X_IR call, int argno, I_IR arg)	Replace argument argno in call with arg.

Table 6: Partial Function Navigation and Modification Interface

Operation	**Description**
X_IR Search(X_IR n, I_IR pattern)	Search for the tree pattern starting at location n. If n is nil, the search starts at the beginning of the function. Unspecified subtrees in pattern can be created using the function I_IR Any(Type ty).
X_IR FindStruct(X_IR n, char *StructName)	Search for use of StructName starting at n.
X_IR FindField(X_IR n, char *StructName, char *FieldName)	Search for use of field FieldName of type StructName starting at n.
Fields *ImportFields(Symbol p, unsigned *n)	Returns a pointer to an array of pointers to data structure p's fields. Elements in this vector can be reordered, deleted, and added.
Fields *ExportFields(Symbol p, Fields *fieldlist, unsigned *n)	Export fields (defined by fieldlist) as the layout for data structure p.
Field AddField(Symbol p, Field f1, Field f2)	Add field f2 after field f1 in structure p.
Field PushField(Symbol p, Field f1, Field f2)	Add field f2 before field f1 in structure p.
Field OverrideField(Symbol p, Field f, Type ty)	Change field structure p's field f type to ty.
Field FirstField(Symbol p)	Returns the first field in data structure p.
Field NextField(Symbol p, Field f)	Returns the next field in data structure p.

Table 7: Partial Structure Navigation and Modification Interface

Code Composition as an Implementation Language for Compilers

James M. Stichnoth and Thomas Gross
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Code composition is an effective technique for a compiler to implement complex high-level operations. The developer (i.e., the language designer or compiler writer) provides building blocks consisting of sequences of code written in, e.g., C, that are combined by a composition system to generate the code for such a high-level operation. The *composition system* can include optimizations not commonly found in compilers; e.g., it can specialize the code sequences based on loop nesting depth or procedure parameters. We describe a composition system, Catacomb, and illustrate its use for mapping array operations onto a parallel system.

1 Introduction

Application-specific programming languages capture properties of a particular problem domain. For many such problem domains, it is not necessary to design and implement a completely new language. Instead, a programming language like C or Fortran serves as the base and is augmented by operators or control constructs to capture specific information about a problem domain. For example, Fortran and C have been extended with array assignment statements and abstractions for computer vision [10, 28], or parallel looping or synchronization constructs have been added. These extensions can usually be expressed in the form of high-level operators. The benefits of such an extension are obvious: the extensions address the concerns of the problem domain, and the general-purpose language can be used for everything else.

There are two challenges in implementing such high-level operators. First, the implementation of such extensions in the compiler must be cheap, with respect to the time and effort that is required. Otherwise, the language may never be used due to the lack of a decent implementation. The second challenge is to devise a high-quality implementation. There are many dimensions of quality, but the two most critical are correctness and efficiency. While correctness is crucial for every language translator, one aspect of correctness that has sometimes been overlooked in the past is completeness; i.e., the translator must be able to deal with all possible legal inputs, and the implementor must cover and test all the boundary conditions. Efficiency is also important, since one of the reasons an application-specific language is used is that such a language can produce better-performing code. Unfortunately, these two demands (correctness and efficiency) tend to increase the cost of implementation, so there is increased interest in techniques to address these problems.

A straightforward implementation of such language extensions is to provide a subroutine for each of the new operators. Since these operators are at a high level and provide powerful operations, the implementation of such a library is far from simple. Furthermore, using a library either deprives the system of opportunities to optimize the code (if the library handles only the general case) or results in many variants (for different parameter values).

A composition system offers an attractive alternative. The developer of the high-level operations provides code sequences that are "stitched together" by a composition system. An analogy to conventional compilers may illustrate the concept. A compiler takes assembly-language or intermediate-code sequences, which have been determined as code for a statement or operator, and optimizes these low-level code sequences. The compiler discovers redundant operations and manages resources (e.g., registers) across boundaries. A composition system takes blocks of code in some suitable high-level general-purpose language and composes the code for the high-level operation by combining and optimizing the code sequences. Since the composition system sees a global view of the program, it can optimize these code sequences better than a conventional compiler.

In a conventional compiler, the compiler developer decides how the assembly language or intermediate code sequences are selected and optimized. However, to support a wide range of high-level operations in a composition system, the developer of the application-specific high-level operations must be able to express a wide variety of actions by the composition system. Thus the code composition must be programmable; i.e., there must be a programming language to control code composition. This programmability provided by the composition language is the key to the power of the composition system. In the remainder of the paper, we first provide a few examples of what we call "complex high-level operations" from different problem domains. Then we discuss code composition as a technique to address the two challenges mentioned above. Then we describe the Catacomb system that has been implemented and supports our claim of the practicality of the idea of code composition. There are other approaches to address the problem, and we summarize those after we present an evaluation of the Catacomb system for one class of high-level operations.

2 Examples of complex high-level operations

This work is motivated by the challenges faced in compiling "complex high-level operations." In this section, we briefly present three examples of complex high-level operations. They are described in more detail in Section 5.

2.1 Array assignment statement

The array assignment statement, which is a key component of High Performance Fortran (HPF) [10], effects a parallel transfer of a sequence of elements from a source array into a destination array. The canonical form of the array assignment statement is

$$A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B].$$

The *subscript triplet* notation $\ell\colon h\colon s$, also used in Fortran 90, describes a sequence of array indices starting with ℓ, with stride s, and having an upper bound h.
An important aspect of the array assignment statement is that the array elements are distributed across the processors of a multiprocessor system. This data distribution dramatically increases the complexity of an efficient algorithm to execute the statement.
Compiling the array assignment statement becomes much more difficult when we extend the canonical case to the general case. There are three extensions: multidimensional references (e.g, $A[1\colon m][1\colon n]$), multiple right-hand side terms (e.g., $B[\ell_B\colon h_B\colon s_B] + C[\ell_C\colon h_C\colon s_C]$), and a mix of subscript triplets and scalar indices (e.g., $A[x][\ell_A\colon h_A\colon s_A]$). While it is conceptually simple to lift the restrictions on the canonical example, it is much more difficult in practice to implement the general case within a compiler or runtime library framework. As such, most proposed implementations ignore the engineering issues of embedding general array assignment algorithms into a parallelizing compiler.

2.2 Data transfer in irregular applications

The array assignment statement is used to perform *regular* data transfer, in which the access pattern (as specified by the subscript triplet) and the data distribution both have regularity that can be exploited at compile time. When the data distribution becomes *irregular*, or the access pattern becomes irregular, the data transfer depends on values available only at run time. Irregular access patterns result from multiple levels of array indirection; e.g., $A[IA[\ell\colon h\colon s]]$. Irregular access patterns also result from explicit loops containing multiple levels of indirection.
Irregular array assignment statements and irregular parallel loops are examples of complex high-level operations. Engineering difficulties arise in a compiler when we add multiple levels of indirection, and when we distribute several dimensions of a multidimensional array.

2.3 Archimedes

The Quake project [4] at Carnegie Mellon focuses on predicting the ground motion during large earthquakes. At its heart is Archimedes [19], a system for compiling and executing unstructured finite element simulations on parallel computers. The compiler component of Archimedes, called Author, presents the programmer with a language nearly identical to C. The language is enhanced with additional aggregate data types for nodes, edges, and elements of a finite element mesh, as well as statements for iterating in parallel over the collection of such objects in the mesh (e.g., FORNODE and FORELEM). In addition, Author provides a crude mechanism that allows the programmer to extend the system with macro-like constructs that support type-checking of their arguments.
The statements that iterate in parallel over the aggregate data types are examples of complex high-level operations. In addition, there are high-level constructs that result in both regular and irregular communication, as described above.

3 Code composition

Traditionally, the compiler translates each input operation or statement into a small fixed sequence of statements at a lower level of abstraction (e.g., machine instructions or operations in an intermediate representation). For each input construct, there is typically a small sequence of instructions to perform the task at execution time. This compilation strategy is called *custom code generation*.

As a high-level language grows to be more complex, the complexity of individual operations increases as well, requiring more and more low-level operations to implement each complex input construct. At this point, the typical approach is to shift to the *runtime library routine* compilation strategy. In this strategy, the compiler translates each such high-level operation into a call to a runtime library routine. This approach tends to be much more manageable than generating custom code, because the code appears in a straightforward fashion in the library, rather than being buried in the compiler. However, it also tends to suffer in terms of runtime performance, because the runtime library routine does not have access to the specific parameters of the construct that are known at compile time, and cannot optimize accordingly.

The compilation strategies of custom code generation and runtime library routines trade off three important issues: efficiency, maintainability, and generality. Efficiency refers to the performance of the generated code at run time; i.e., being able to optimize the construct's runtime execution, based on all available compile-time information. Maintainability refers to how easy and straightforward it is to develop and maintain the algorithm, within the framework of the compilation system. Generality refers to whether the general case or merely a simplified canonical case is implemented.

Our solution is a technique called *high-level code composition*. Code composition is an approach related to custom code generation. The code sequences to be produced, called *code constructs*, appear external to the compiler. The instructions for piecing together the code sequences also appear externally, rather than being embedded in the compiler. These instructions are called *control constructs*. Code constructs and control constructs are bundled together into manageable-sized chunks called *code templates*, in the same way that the code in a typical program is a collection of manageable-sized functions. The code templates form a specialized language for directing the compilation process.

There is a *composition system* coupled with the compiler that uses these code templates to produce code. Figure 1 shows how the composition fits in with respect to the rest of the compiler. This structure is important for code reuse: the same composition system can be used in several different compilers, for several different problem domain. The composition system's function is to *execute* the code templates at compile time. Executing the code templates means following the instructions specified by the control constructs. Because the control constructs constitute a programming language, the composition system can be thought of as an interpreter of the control constructs.

The composition system is invoked by the compiler, which instructs the system to execute a template on a particular input. This input is a high-level programming language operation, such as an array assignment statement. The composition system then executes the template with full knowledge of all compile-time information (for the array assignment, information like number of array dimensions, dimension sizes, and distribution parameters). It uses this knowledge to produce the correct code for the input and to optimize the code.

The control constructs constitute a small language that the composition system interprets at compile time. As such, they must be designed as one would design a real programming language, containing variables, conditionals, procedure calls, and so forth. Furthermore, it is important to choose a syntax that is easily distinguishable from the syntax of the code constructs. There are several features that should be present in the control constructs: control procedures and procedure calls, control variables, control variable assignment, a control test, a control loop, and a concept called *variable renaming*. All of these features but the last are fairly self-explanatory. Variable renaming is a subtle point that is easy to overlook, but is important in practice. Often we need to compose the same template several times, but each composition needs a different set of variable names. For example, we might want to recursively compose a basic "loop" template several times to create a loop nest, but each loop induction variable name must be unique. To allow this, a variable renaming operator allows new variable names to be constructed during template execution, similar to Lisp's `gensym` function, only more controllable.

There is also the question of what, if any, relationship the control and code constructs should have toward each other. There are two styles in which the constructs can be interleaved: the *syntactic* style and the *lexical* style. With the syntactic style, code constructs and control constructs are required to fully nest within each other. With the lexical style, there is no such requirement.

Figures 2 and 3 demonstrate these two styles for writing templates, assuming C-like control constructs and Fortran-like code constructs. The purpose of the code in the figure is to produce an n-deep loop nest, where n is a parameter passed to the template. The template would be invoked through the control construct `call_template(loopnest, n)`. The lexical style uses a control loop to generate the `DO` statements and the matching `END DO` statements. The syntactic style has to use a recursive template to form a loop nest, calling itself recursively between the `DO` and the `END DO`.

It is preferable for the code and control constructs to interact through the syntactic style. The primary benefit of the syntactic style is readability of the template code.

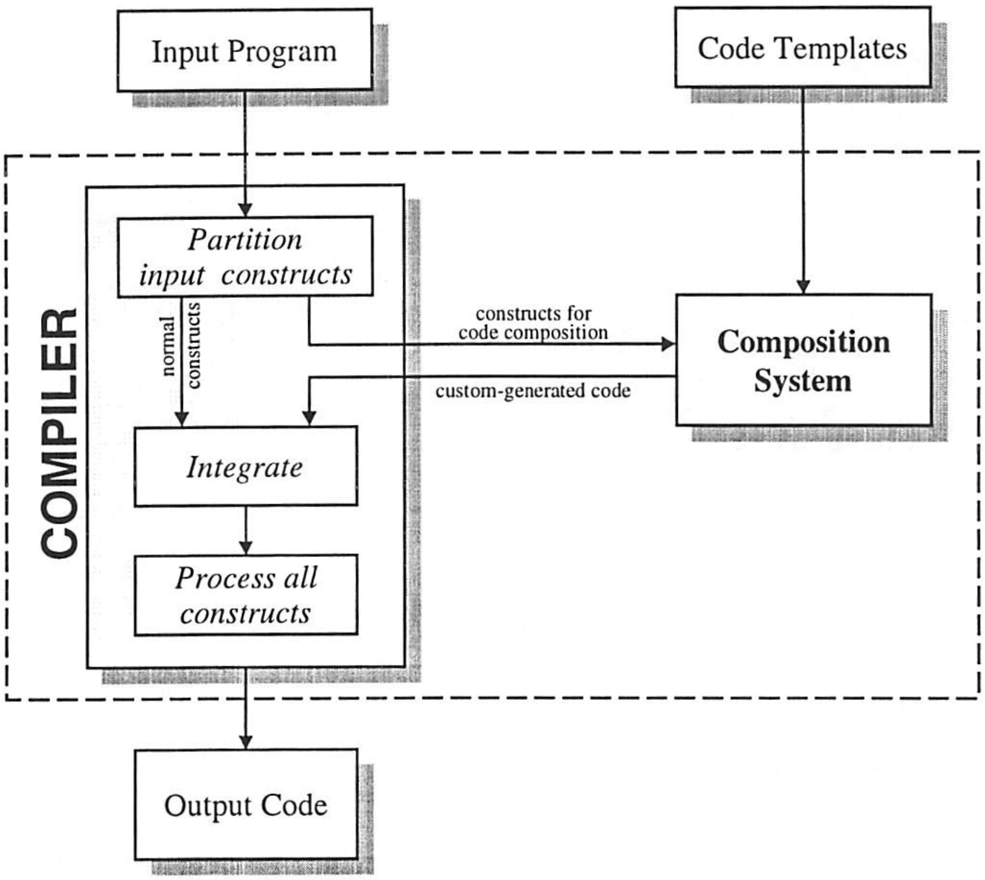


Figure 1: Integration of a composition system into a compiler.

```
TEMPLATE loopnest(depth)
{
  call_template(loopnest1, 0, depth);
}

TEMPLATE loopnest1(cur_depth, max_depth)
{
  if (cur_depth < max_depth) {
    DO I(cur_depth) = 1, 10
      call_template(loopnest1, cur_depth+1, max_depth);
    END DO
  } else {
    /* inner loop code goes here */
  }
}
```

Figure 2: A template for constructing a loop nest, using the *syntactic* style.

```
TEMPLATE loopnest(depth)
{
  for (count=0; count<depth; count++) {
    DO I(count) = 1, 10
  }
    /* inner loop code goes here */
  for (count=depth-1; count>=0; count--) {
    END DO
  }
}
```

Figure 3: A template for constructing a loop nest, using the *lexical* style.

It is easier to develop and maintain code written with this style, and it is also easier to automatically detect syntactic mistakes in the template code. Using the lexical style, it is much easier to make mistakes in matching up the syntactic constructs in the generated code (e.g., the DO and END DO in a Fortran loop). But with the syntactic style, syntax errors are obvious when the code or control constructs do not match up correctly, and can be easily detected when the composition system parses the templates.

It is important to stress that a composition language is *not* meant to be programmed by the end user. Rather, the composition system is a tool used by the *compiler writer* to facilitate the translation of complex high-level operations into lower-level code.

4 Catacomb

To illustrate the usefulness of a composition system, we have developed Catacomb, a composition system for generating C code. As such, its code constructs have the syntax and semantics of C constructs. The control constructs are C-like, but since they interleave with the code constructs using the syntactic style, the syntax is slightly different from C.

To illustrate the combination of code and control constructs, Figure 4 shows an annotated set of Catacomb templates. Its purpose is to construct a loop nest for setting the elements of a multidimensional array. Below the set of templates is a sample invocation and its corresponding result. The example consists of three templates:

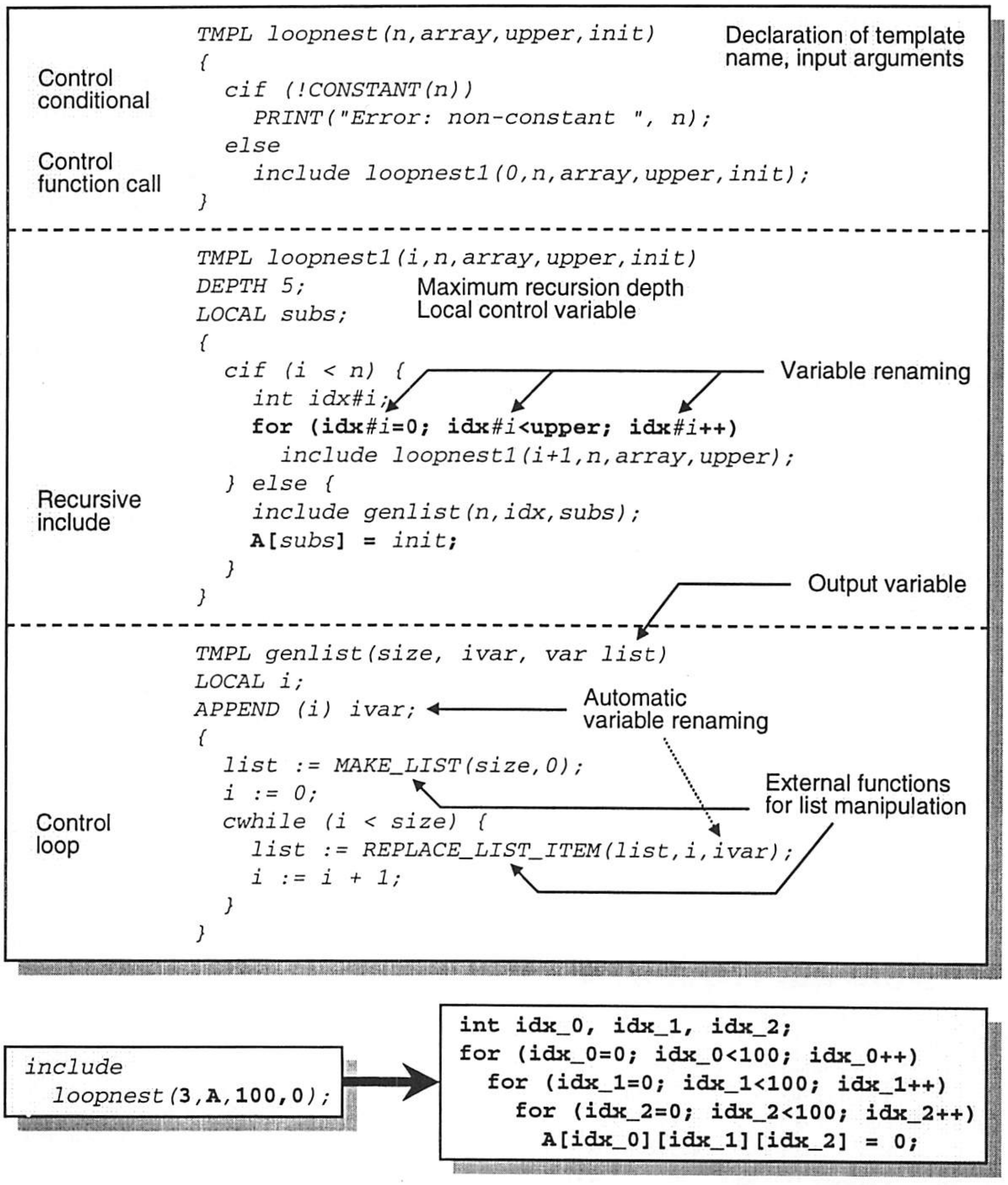
- `loopnest`: The entry point. It takes three input arguments: the number of array dimensions (equivalent to the depth of the loop nest to be produced), the array whose elements are to be set, and the size of the array dimensions (for the sake of simplicity, all dimension sizes are assumed to be equal). The template verifies that `n`, the number of array dimensions, is a compile-time constant, and then calls the recursive `loopnest1` template.

- `loopnest1`: The recursive template. This template generates the outer loop, and then calls itself recursively to generate the rest of the loop nest. When it reaches the innermost loop, it generates the inner-loop assignment statement.

- `genlist`: Creation of the array subscript list. Because the number of array dimensions is an input parameter to the `loopnest` template, the subscript list for the inner-loop array reference has to be generated each time the template is called. The `genlist` template is responsible for building the subscript list.

This set of templates illustrates all of Catacomb's control constructs. Also in the figure is a sample invocation of the entry template, and the resulting C code. This is a complete working example of a Catacomb template and its output, with the caveat that the actual declaration of the input array is omitted; the array is assumed to be declared elsewhere.

The template header declares the template and its arguments, as well as template-local control variables, syntactically similar to a C function declaration. Control variables can be set using the `:=` control assignment operator. The `include` statement executes a control function call, passing a list of arguments to a template. By default, arguments are passed by value; the `var` keyword in the template argument declarations allows arguments to be passed by value-result, making it possible for a template to return results to the caller. The `cif` and `cwhile` are the control conditional and control loop, respectively; the condition must evaluate to a compile-time constant.



```
TMPL loopnest(n,array,upper,init)
{
  cif (!CONSTANT(n))
    PRINT("Error: non-constant ", n);
  else
    include loopnest1(0,n,array,upper,init);
}
```

```
TMPL loopnest1(i,n,array,upper,init)
DEPTH 5;
LOCAL subs;
{
  cif (i < n) {
    int idx#i;
    for (idx#i=0; idx#i<upper; idx#i++)
      include loopnest1(i+1,n,array,upper);
  } else {
    include genlist(n,idx,subs);
    A[subs] = init;
  }
}
```

```
TMPL genlist(size, ivar, var list)
LOCAL i;
APPEND (i) ivar;
{
  list := MAKE_LIST(size,0);
  i := 0;
  cwhile (i < size) {
    list := REPLACE_LIST_ITEM(list,i,ivar);
    i := i + 1;
  }
}
```

```
include
  loopnest(3,A,100,0);
```

```
int idx_0, idx_1, idx_2;
for (idx_0=0; idx_0<100; idx_0++)
  for (idx_1=0; idx_1<100; idx_1++)
    for (idx_2=0; idx_2<100; idx_2++)
      A[idx_0][idx_1][idx_2] = 0;
```

Figure 4: A set of Catacomb templates for constructing a loop nest that sets values in a multidimensional array. Also depicted are a sample invocation and the corresponding result.

Catacomb introduces the # operator for variable renaming. For example, `x#3` evaluates to the variable `x_3`, `y#4#5` evaluates to the variable `y_4_5`, and `foo#"bar"` evaluates to the variable `foobar`. (When the right operand is an integer constant, Catacomb also inserts an underscore character, "_", to help avoid variable name conflicts.) To aid in the conversion of library routines into templates, Catacomb provides the `APPEND` statement in the template header. With the statement `APPEND (i) x;` in the header, every occurrence of the variable `x` in the template body is automatically replaced with `x#i`.

In addition, Catacomb provides a number of control functions, called *external functions*, to perform operations not possible using the basic C operators on which the control constructs are based. For example, `CONSTANT` is used to test whether the input evaluates to a compile-time constant. There is a default set of external functions in Catacomb, and the set is easily extended (e.g., to add functions that query the distribution parameters of HPF arrays).

Catacomb implements several global optimizations [1, 8], as well as some nonstandard optimizations based on *bounds analysis*. Bounds analysis is based on the observation that sometimes, even though the compiler cannot determine a specific value for a variable or expression, it can determine that it must fall within a certain range of values. Catacomb uses bounds information to simplify expressions where possible. It uses copy propagation techniques to propagate the bounds across assignment statements. Catacomb also extracts bounds information from `if` conditions where possible. Because standard C optimizers do not implement bounds analysis, and some bounds information is available only within Catacomb (e.g., the number of processors in an array distribution is always positive), Catacomb needs to include these optimizations.

There is an additional issue related to the Catacomb implementation: the execution model of the interaction between control constructs and global optimizations. The most obvious and straightforward way to integrate them is to make them completely independent. This suggests a two-phase execution approach: in the first phase, Catacomb executes only the control constructs, leaving just the code constructs, and in the second phase, the resulting code constructs are passed off to the global optimizer, and then emitted. An attractive feature of the two-phase approach is the simplicity in both semantics and implementation.

Unfortunately, under this model, code composition decisions can only be made based on the values of control variables. For example, consider the following template code:

```
        x=1; cif (x==1) { ...  }
```

Note that `x` is a code variable, not a control variable. Under the two-phase execution model, even though it is obvious that the value of `x` is always 1 at the `cif` site, the control constructs are executed before the global optimizations, and thus the `cif` has no way of knowing that the value of `x` is 1 at that point.

There are several alternative template programming styles, execution models, and semantics for allowing composition decisions to be made based on the values of code variables (these are discussed in greater detail elsewhere [20]). Most are insufficient and/or have unacceptably confusing semantics under different circumstances. The best alternative, although naturally the most difficult to implement, is a single-phase execution model, in which global optimizations are performed at the same time as the control execution. The single-phase model improves the efficiency of the generated code; the implementation details are beyond the scope of this paper.

5 Complex high-level operations

In this section, we consider two examples of complex high-level operations, and demonstrate why code composition is superior to the standard compilation techniques for these operations.

5.1 Array assignment statement

The array assignment statement is nontrivial to execute because of two properties: the elements of the arrays are distributed across different processors, and the sequence of elements indexing each array can be an arbitrary arithmetic sequence. Evidence of the complexity and importance of the array assignment statement can be found in the number of different algorithms that have been proposed for executing it efficiently [21, 11, 7, 13, 26, 14, 2, 25]. The compact syntax that hides the complexity of an efficient and complete implementation is what makes the array assignment statement particularly interesting.

The canonical form of the array assignment statement is

$$A[\ell_A : h_A : s_A] = B[\ell_B : h_B : s_B].$$

The statement is equivalent to the pair of sequential loops shown in Figure 5.

HPF arrays are allowed to have a *block-cyclic* distribution, in which fixed-size blocks of array elements are distributed to the processors in a round-robin fashion (see Figure 6). Two parameters characterize this distribution: the block size b and the number of processors P. The distribution defines an *ownership set* for each processor, which is the set of array elements mapped to

```
j = $\ell_B$
DO i = $\ell_A$, $h_A$, $s_A$
  $T$[i] = $B$[j]
  j = j + $s_B$
END DO
-----------------
DO i = $\ell_A$, $h_A$, $s_A$
  $A$[i] = $T$[i]
END DO
```

Figure 5: Sample sequential code for the array assignment statement $A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B]$.

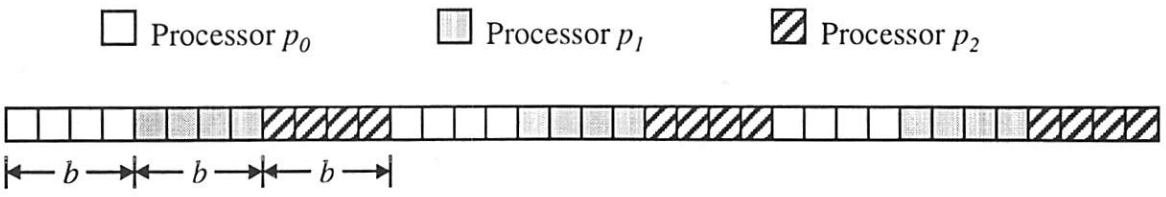


Figure 6: A block-cyclic data distribution in HPF, with three processors ($P = 3$) and block size $b = 4$.

the processor. Extremal cases of the block-cyclic distribution include the block distribution, in which a single, suitably large block is distributed onto each processor, and the cyclic distribution, in which the block size is 1.
For the canonical array assignment statement, the challenge is to efficiently enumerate the *communication set*, the set of array elements to be transferred between a given pair of processors. The subscript triplets and the two processors' ownership sets determine the communication set. Implementations of the published algorithms for the array assignment require on the order of hundreds of lines of C code. For this reason, the array assignment is considered to be a complex high-level operation.
Compiling the array assignment statement becomes much more difficult when we extend the canonical case to the general case. There are three extensions: multiple right-hand side terms (e.g., $B[\ell_B\colon h_B\colon s_B]+C[\ell_C\colon h_C\colon s_C]$), multidimensional references (e.g, $A[1\colon m][1\colon n]$), and a mix of subscript triplets and scalar indices (e.g., $A[x][\ell_A\colon h_A\colon s_A]$). While it is conceptually simple to lift the restrictions on the canonical example, it is much more difficult in practice to implement the general case within a compiler framework.
Embedding an array assignment algorithm into a compiler severely degrades the *maintainability* of the algorithm. It becomes difficult to write the code within the compiler framework, and equally difficult to read and modify the code. For this reason, it is tempting to use the runtime library routine approach, writing the array assignment algorithm in a straightforward fashion in a runtime library.
Unfortunately, efficiency and generality suffer under the runtime library routine approach. Efficiency suffers because in general, it is no longer possible to compile all parameters that are known at compile time into the library. For the canonical array assignment statement, there are 14 parameters: 3 parameters for a subscript triplet, and 4 parameters for the distribution (block size, number of processors, array size, and processor number), all of which is multiplied by two because the statement operates on two arrays. Usually, most of these parameters are known at compile time, allowing more efficient code to be generated.
Generality suffers in a runtime library as we extend the canonical array assignment to the general case. This generality has three aspects: multidimensional arrays, multiple right-hand side terms, and scalar subscripts (as opposed to subscript triplets). Implementing the array assignment for multidimensional arrays requires iterating over a loop nest whose depth is proportional to the number of dimensions. There are two ways to implement such a loop nest in a runtime library. The first is to write a separate routine for each dimensionality, but this method imposes a limit on how many dimensions can be handled in an array, and it creates an exponential code explosion in the amount of code to write in the library. The other method is to write a function that recursively calls itself for each successive level of the loop nesting. However, this method lowers runtime efficiency by precluding function inlining and by imposing additional overhead in the inner loop.
Code composition is easily used to handle multidimen-

sional arrays. Using a recursive template like that of Figure 4, the composition system generates a loop nest customized for the specific input array assignment statement. There are then no limits on the number of dimensions, nor is there extra inner-loop overhead. In addition, there are no exponential code explosion problems reducing maintainability.

A runtime library routine that handles multiple right-hand side terms is even more difficult to write than one for multidimensional arrays. Such a library must be able to handle an arbitrary set of arithmetic operators describing the right-hand side expression. Without runtime code generation techniques, the inner loop in which the computation is performed is bound to have a great deal of overhead. On the other hand, using code composition, the code templates create a customized inner loop, into which the specific operators are compiled. Once again, the increased generality does not cause any additional inner-loop overhead.

Although scalar subscripts generally do not cause efficiency problems in a runtime library, handling them generally decreases the maintainability of the runtime library. In fact, as each aspect of generality is added to a runtime library, there is a corresponding decrease in maintainability of the library. Code composition, on the other hand, allows the templates to directly manipulate subscript lists at compile time, providing generality with little impact on maintainability.

5.2 Data transfer in irregular applications

The array assignment statement is used to perform *regular* data transfer. The regularity of the transfer arises from the regularity of the subscript triplet and the block-cyclic distribution. Sparse and unstructured problems result in *irregular* data transfer, due to irregular data distributions and irregular access patterns (e.g., array references with multiple levels of indirection). In an irregular problem, the data transfer depends on values available only at run time; thus runtime analysis is required.

The key to the runtime analysis is the *inspector/executor* approach. This approach divides the runtime execution into two parts, the inspector phase and the executor phase. The inspector analyzes the global access pattern and calculates which array elements each processor needs to send, and where to send each element. The executor carries out the data transfer and the computation. The most effective means of executing irregular computations is through the use of the PARTI or CHAOS runtime libraries [24, 17, 18], developed by Saltz et al. These libraries contain sophisticated routines that help the programmer translate a sequential program into a parallel program that uses the inspector/executor approach.

More recently, several parallelizing compilers [6, 30] analyze the sequential loops in a program and automatically produce parallel loops with calls to the appropriate CHAOS routines. For many kinds of simple sequential loops, this translation is fairly straightforward. However, the translation becomes more complex as more levels of indirection in the array references are added.

For compiling irregular programs that use multiple levels of indirection in distributed arrays, Das, Saltz, and von Hanxleden use a technique called *slicing analysis* [9]. The idea behind slicing analysis is that for each loop containing a distributed array reference with multiple levels of indirection, that loop can be rewritten as several loops, each of which contains only a single level of indirection. The resulting program, containing only single levels of indirect array references, is then amenable to parallelizing techniques in existing compilers for irregular problems.

Because of the complex structure of an irregular parallel loop, the loop itself is not amenable to being implemented purely with a runtime library routine. Instead, a compiler is needed to translate the loop into a sequential loop, possibly with calls to a support library like CHAOS. The complexity of the code to be generated usually leads the compiler writer to sacrifice some amount of generality in the solution. For example, many compilers only allow a single level of indirection in array references, and existing compilers only allow a single dimension of an array to be distributed.

6 Evaluation

Along with our implementation of Catacomb, we developed templates to implement several solutions to the array assignment statement. These algorithms are known as the CMU algorithm [22], the OSU algorithm [11], and the LSU algorithm [26]. We divided our implementation into three components: preprocessing, architecture, and algorithm. The algorithm component determines the communication sets and packs/unpacks the communication buffers. The architecture component provides an interface to the architecture-specific communication features, such as the specific method for calling the send, receive, and synchronization primitives. The preprocessing component attempts to simplify the input array assignment statement into a form that is closer to the canonical array assignment statement. This division allows an arbitrary algorithm to be combined with an arbitrary architecture to form a complete implementation.

There are three issues that we can evaluate: efficiency, generality, and maintainability. Regarding efficiency, Catacomb's aggressive optimization framework results in code whose quality is close to that of hand-tuned code. Because this paper focuses more on the software engineering issues, we omit a discussion of the performance of the generated code; details are available elsewhere [20].

Regarding generality, the code templates make it simple and straightforward to support any regular array assignment statement, containing an arbitrary number of dimensions, and arbitrary number of right-hand side terms, and an arbitrary interleaving of scalar subscripts and subscript triplets. None of the implementations of the algorithms we studied handle more than the canonical case, yet the Catacomb template mechanism enables us to automatically extend the algorithm for the canonical case to the general case. In fact, this is the first implementation of the array assignment that allows an arbitrary algorithm to be coupled with an arbitrary architecture to form a complete implementation.

Evaluating maintainability is largely a subjective task. There is, however, an objective measurement that gives a rough idea of the maintainability of our template framework for the array assignment. This measurement consists of looking at the breakdown of the template code into control and code constructs, and comparing the amount of code constructs to the amount of control constructs. One could argue that as the amount of control constructs increases, the actual code being produced (i.e., the code constructs) becomes increasingly obscured within the control constructs, and the maintainability correspondingly decreases.

Figure 7 shows the breakdown of the CMU, OSU, and LSU template code, as well as the MPI communication architecture template code. This measurement was taken after removing comments and blank lines from the templates, and should only be considered as an approximation. We consider the number of lines of control constructs, code constructs, and external support libraries (i.e., code from the original implementation that did not need to be converted to template code). The breakdown shows that the control constructs are relatively evenly matched with the code constructs. In contrast, the original implementation of the CMU algorithm in the Fx parallelizing compiler [23] required roughly 15,000 lines of compiler code. Given that the CMU algorithm itself contains less than 1,000 lines of code, we can see that the vast majority of the 15,000 lines was dedicated to compile-time control. Because the Catacomb templates contain far less code devoted to compile-time control, the implementation is far more maintainable.

7 Related work

7.1 Templates and macro processing

Code composition includes control constructs that allow generalized computation at compile time. The concept of compile-time compilation has been around for some time. A widely-used example today is the C preprocessor. Its computational power is extremely limited, though; for example, looping is not possible, either directly or through recursion. Furthermore, its decoupling from the compiler prevents anything like the single-phase integrated execution model mentioned in this document at the end of Section 4. A consequence that many C programmers may be familiar with is the inability to perform preprocessor operations like `#if sizeof(int)==4`. PL/I [15] offers a more powerful preprocessor. However, it also is incapable of a single-phase execution mode, and neither it nor the C preprocessor is equipped to perform structural queries on general expressions, a feature critical to code composition.

The C++ template system provides a simple way to generate new functions and methods, tailored to a specific data type. Veldhuizen [27] has developed a mechanism called *expression templates*, which allows the template system to compose code in more complex ways, based on the structure of input expressions. For example, with the appropriate declaration of `x` and definition of `integrate`, the statement `double result = integrate(x/(1.0+x), 0.0, 10.0);` produces custom code at compile time to integrate the function $x/(1+x)$ over the domain $0 \leq x \leq 10$. While this is an interesting way to gain compile-time control over the structure of an expression, in practice the specifications end up being overly complex and unreadable.

There are other macro extensions to C (e.g., Safer_C [16] and Programmable Syntax Macros [29]) that offer many of the same benefits as Catacomb. These systems are generally not extensible like Catacomb, and do not offer an integrated single-phase execution model, thus precluding the use of global optimizations in the macro processing decisions.

Barrett et al. [5] use the concept of templates in a numerical computation context. Templates are designed and written in a high-level language to handle specific features of iterative solvers for linear systems (e.g., sparse or dense, convergence requirements, sequential or parallel, data layout). At compile time, the system automatically finds the right set of templates to match the needs of the user. This kind of system fits well within the code composition framework we describe.

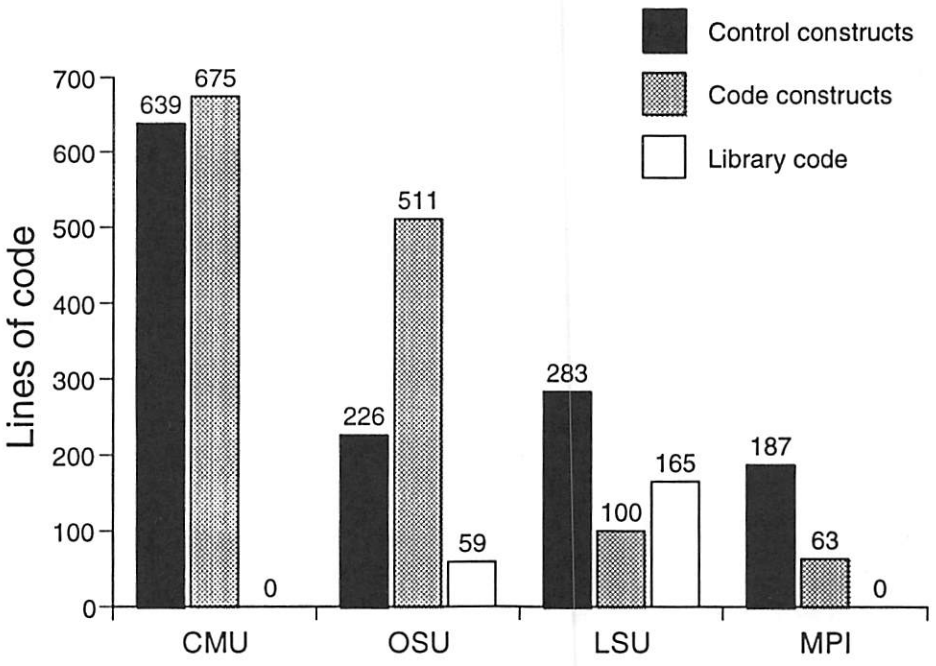


Figure 7: Comparison of the number of lines of control constructs and code constructs in the Catacomb templates for the array assignment, as well as lines of code in support libraries.

7.2 Partial evaluation

Like most optimizing compilation systems, Catacomb and the concept of code composition are related to the field of *partial evaluation* [12]. A partial evaluation system takes as input a program in a source language, and a set of known inputs to the program, and produces a *residual program* specialized for those particular inputs. The code templates are similar to a two-level version of an imperative input language. The control constructs, including the control variables, explicitly have a static binding time. The binding times of the code constructs are analyzed online using the global optimization framework, allowing some variables and statements to be classified as static, rather than the default of dynamic. Catacomb's technique of bounds analysis has some similarity to *bounded static variation*, in which the partial evaluator can restrict an otherwise-dynamic variable to a finite set of static values.

A notable difference between code composition and standard partial evaluation is the fact that control constructs follow a different flow of control from the code constructs. For example, a control assignment statement in the body of a loop is executed exactly once, regardless of the number of loop iterations at run time. This means that the straightforward translation of control constructs into their corresponding code constructs does *not* preserve the original semantics, in contrast to standard partial evaluation. Future work in this direction is to explore the issues of whether the Catacomb model (which is similar to the C preprocessor model) or the standard partial evaluation model presents the user with more "natural" semantics and ease of use, and whether there is in fact a realistic situation in which Catacomb's semantics are necessary.

7.3 Runtime code generation

Dynamic approaches attempt to improve the code at run time, and dynamic methods have lately received renewed attention (e.g., [3]). If the program notices that some parameter always has the same value, a runtime optimizer can customize the program by working with the known parameter values. Since these values are known, some tests may be resolved, or special instructions chosen, and such transformations have the potential to improve performance. However, dynamic methods too face a number of challenges: for one, the system must ensure that the overhead spent on detecting the occurrence of a common scenario is bounded and in relation to the expected benefits.

Runtime code generation should be used as an additional performance enhancement to code composition, rather than as a replacement. Optimizations should be performed in advance by the compiler whenever possible. In addition, using runtime code generation in place of code composition requires a full runtime library for the problem to be designed, which still trades off maintainability and robustness. (Efficiency is ignored, since it would be the responsibility of the runtime code generation system to provide runtime efficiency.)

8 Concluding remarks

We have identified a class of high-level languages formed by adding complex high-level operations to a base language. For these languages, traditional compilation techniques are inadequate. The traditional techniques, namely custom code generation and runtime library routines, fall short because custom code generation offers efficiency and robustness at the cost of maintainability, while runtime library routines offer maintainability at the cost of efficiency and robustness. We developed a new method called code composition and implemented a code composition system to demonstrate the practicality of this idea.

Code composition is under the control of the implementor of the complex operations, who uses a domain-specific language for directing the compilation process. Programmable code composition provides the union of the benefits of the traditional approaches: the composition system optimizes the code and thereby ensures efficiency, yet the composition templates are concise.

We designed and implemented Catacomb, a system for code composition, and explored several issues relating to the automatic optimization of the code it produces. We implemented several algorithms for the HPF array assignment statement in the context of Catacomb, and used the implementation to evaluate several aspects of efficiency, maintainability, and robustness. In our experience, use of a code composition system is a good way to control the translation of complex operations and provides for an elegant and effective approach to producing high quality code without undue implementation cost.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. Technical Report A-278-CRI, Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris, November 1995.

[3] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 1996. ACM.

[4] H. Bao, J. Bielak, O. Ghattas, D.R. O'Hallaron, L.F. Kallivokas, J.R. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Supercomputing '96*, Pittsburgh, Pennsylvania, November 1996.

[5] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, Pennsylvania, 1994.

[6] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[7] S. Chatterjee, J. Gilbert, F.J.E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[8] F. Chow. *A Portable Machine-Independent Global Optimizer – Design and Measurements.* PhD thesis, Stanford University, 1984.

[9] R. Das, J. Saltz, and R. von Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 152–168, Portland, Oregon, August 1993. Springer Verlag.

[10] High Performance Fortran Forum. High Performance Fortran language specification version 1.0, May 1993.

[11] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32(2):155–172, February 1996.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall, 1993.

[13] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings*

of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 102–111, Santa Barbara, California, July 1995.

[14] S. Midkiff. Local iteration set computation for block-cyclic distributions. Technical Report RC-19910, IBM T.J. Watson Research Center, January 1995.

[15] S. Pollack and T. Sterling. *A Guide to PL/I and Structured Programming (Third edition).* Holt, Rinehart, and Winston, New York, 1980.

[16] D.J. Salomon. Using partial evaluation in support of portability, reusability, and maintainability. In Tibor Gyimóthy, editor, *Proceeding of the Sixth International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 208–222, Linköping, Sweden, April 1996. Springer Verlag.

[17] J. Saltz, R. Ponnusamy, S.D. Sharma, B. Moon, Y.-S. Hwang, M. Uyasl, and R. Das. A manual for the CHAOS runtime library. Technical Report CS-TR-3437, Department of Computer Science, University of Maryland, March 1995.

[18] S.D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing '94*, pages 97–106, Washington, DC, November 1994.

[19] J.R. Shewchuk and O. Ghattas. A compiler for parallel finite element methods with domain-decomposed unstructured meshes. In David E. Keyes and Jinchao Xu, editors, *Proceedings of the Seventh International Conference on Domain Decomposition Methods in Scientific and Engineering Computing*, volume 180 of *Contemporary Mathematics*, pages 445–450. American Mathematical Society, October 1993.

[20] J. Stichnoth. *Generating Code for High-Level Operations through Code Composition.* PhD thesis, School of Computer Science, Carnegie Mellon University, August 1997.

[21] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[22] J.M. Stichnoth. Efficient compilation of array statements for private memory systems. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[23] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, California, May 1993.

[24] A. Sussman, G. Agrawal, and J. Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1, University of Maryland, December 1993.

[25] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 309–316, Knoxville, Tennessee, May 1994.

[26] A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data-parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing*, 38(2):188–203, November 1996.

[27] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[28] J. Webb. Steps toward architecture-independent image processing. *IEEE Computer Magazine*, 25(2):21–31, February 1992.

[29] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 156–165, Albuquerque, New Mexico, June 1993. ACM.

[30] H. Zima, P. Brezany, B. Chapman, P. Mehrota, and A. Schwald. Vienna Fortran – a language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.

BDL: A Language to Control the Behavior of Concurrent Objects

Frédéric BERTRAND Michel AUGERAUD

Laboratoire d'Informatique et d'Imagerie Industrielle
Université de La Rochelle
Avenue Marillac, 17042 La Rochelle, Cedex 01, France

fbertran@mail.univ-lr.fr maugerau@mail.univ-lr.fr

Abstract

Combining concurrency and object orientation is still difficult. In an approach where methods are concurrency units, one of the main difficulties is the control of the behavior of objects.

Our proposal is BDL a language allowing to express and to achieve this control. We propose a model where each object includes a so called "execution controller" programmed with BDL. This introduces a conceptually clean separation between processing (method execution) and control. The controller ensures the respect of scheduling constraints between the executions of methods. Similarly the behavior of aggregate objects can be controlled. This language has a convenient formal base. Thus, using the expression of control, behavioral properties of an object, or even of a group of interesting objects can be verified. Our approach allows, for example, deadlock detection or verification of safety properties while the compiled object controller keeps a reasonable size.

A compiler has been implemented allowing to automatically generate the controller code from a BDL program. This compilation is achieved by producing an Esterel (reactive programming language) code from a BDL program, the Esterel compiler giving the executable code. Inter-method concurrency is implemented using lightweight processes.

1 Introduction

Through a set of mechanisms (inheritance, aggregation, prototyping) the object-oriented approach is well suited to describe complex systems. The designers of such systems have been aware of naturally concurrent applications existing (such as booking systems) which cannot be easily described with a sequential programming language [26]. So the notion of concurrent object-oriented programming has appeared naturally.

However concurrent programming is still problematic [20, p. 56]. The main problem concerns the control of concurrency, also called synchronization. This control aims at preventing the object from being in an incoherent state as, for example, when two methods concurrently executed modify attributes. A schedule of method executions is necessary to ensure that such cases do not occur.

In many concurrent object-oriented languages, the execution of methods is controlled by guards checked at runtime. This approach presents the disadvantage of mix control code and processing code. We have proposed [2, 6] an architecture for objects dissociating processings offered by the object (achieved by methods) and the control of these processings. This model gathers these conditions in a dedicated structure ensuring, for each request, that the execution is possibly related to the state of execution of the other methods. This construction improves the maintainability of the object by centralizing the execution conditions of each method in only one entity (called *controller*). To express the execution conditions of methods and to program this controller, we have developed a language named BDL (Behavioral Description Language). This paper shows how BDL could be used to express the control of the behavior of simple objects or group of objects. We also describe the BDL implementation based on a reactive language that allows us to verify properties on BDL programs.

In section 2, we explain the need of control for a concurrent object and we present the execution controller achieving this control. The BDL language, which allows to program this controller, is described in section 3. The semantics and the implementation of BDL are detailed in section 4. In section 5, we show how the verification could be achieved on an object or on a group of objects. Related work is described in section 6. Finally, we conclude in section 7 by emphasizing the outlooks offered by this new approach.

2 The Control Of Object Behavior

In this section, we explain why a concurrent object needs an execution control and we describe the object model in which the control is exercised.

2.1 The Need of Control for a Concurrent Object

Inside a concurrent object the method execution may depend on whether other methods are or not active. For example, if an object has two methods `read` and `write`, it is easy to understand that these methods cannot be executed at same time. Their executions must be mutually exclusive. This is a constraint of scheduling.

We will illustrate this type of constraint using a `File` object. The object owns four methods: `open`, `read`, `close` and `write`. There are different constraints on the execution of these methods. Let us suppose first that the object is in only read mode (`write` method is not accessible). In this case, there are two sorts of constraints: a sequentiality constraint between the three first methods and a repetition constraint between the former and the later since the object must be permanently able to process execution requests.

To express these constraints, we have defined a set of operators representing the BDL language (see 3.1). BDL expressions use method names and operators. For example, the BDL expression specifying that instances have to execute repeatedly (indicated by `*` operator) the sequence (indicated by `;` operator) of methods `open`, `read` and `close` in this order is:

```
(open ; read ; close)*
```

This specification may be refined by allowing multiple executions of `read`:

```
(open ; read* ; close)*
```

Now if we consider the file in read-write mode, we can specify an exclusive execution (operator "|") between `read` and `write`:

```
(open ; (write | read)* ; close)*
```

Our work aims to achieve scheduling constraints between methods. Before describing our approach, we will define the concurrent object model upon which the control is defined.

2.2 The Concurrent Object Model

Concurrent object-oriented languages may be classified according to three criterias [22]:

- *the object model* defining the relationship between the structures of execution (*threads*) and the object paradigm;
- *the intra-object concurrency* concerning the management of threads (number of threads, ways of creating and switching threads);
- *the interaction between objects* describing mechanisms that allow to specify the sending and the receiving of messages by objects.

For each of these criterias, our choices have been selected with the following aims:

1. defining the object as a self contained entity possessing its own executive structures;
2. designing an application as a set of communicating distributed objects;
3. improving the capacity of reaction of the object; this is achieved in two ways. Raising the degree of concurrency and therefore allowing a request to be satisfied as soon as the state of the object enables it. Moreover we give to the object a capacity to control method executions (possibility of interrupting or cancelling the execution).
4. possibility to verify the object behavior.

When describing the behavior, we imply the set of the methods of the object. Controlling the object behavior consists, in determining according to the state of the object, whether the execution of a method is authorized and, in this case, to launch this execution. If the execution is not possible, the request may be stored or rejected following a determined policy.

We have chosen an active object model. This model has been adopted by a great number of languages on concurrent object-oriented systems such as Pool-T [1], Eiffel// [10] and Rosette [27]. An active object decides, according to the object state, the time when a method execution can be run. The object has a part of both a client (when it requires the execution of a method of another object) and a server (when it executes one of its methods on the request from another object). Furthermore, this model is well adapted to the fourth aim: the verification of the behavior control is easier to achieve if the control is carried out by the object and not an external structure because the object is a self-contained entity.

In order to rise the capacity of service, an inter-method concurrency model has been adopted. So every method execution takes place in a thread of execution.

Finally the need of verification of some properties (safety, liveness) on the behavior control of the object is made easier using a centralized control achieved by a dedicated structure on which the verifications are carried out. These properties may be, for example, deadlock freeness, the respect of mutual exclusion during the executions of methods changing the object state, or still the respect of the sequentiality constraints between these executions. The previous example of `File` object has shown that scheduling constraints must be respected. We call the entity in charge of this control, the *execution controller*. Furthermore, this centralized control allows a better reuse by a clear separation between control and processing.

2.3 The Execution Controller

The controller, depicted by figure 1, has three functionalities. The first one is related to the execution management. The controller carries out the creation of the thread for every method having to be executed and then associates code and thread for the execution.

The second one, more complex, consists of synchronizing the execution according to the BDL program. To achieve this functionality, the controller must, permanently, be aware of the execution state of methods. At this time, the model of execution controller is restricted: the controller does not use object attributes to manage the executions; the activation conditions of a method that use attributes are checked in method code. This restriction is necessary to use verification tools such as FcTools which works using finite transition systems.

In fact this restriction avoids introducing numerical data in the controller. Because for checking transition systems tools use finite structure thus the use of numerical variables with infinite domains is not possible.

The third one concerns the storage of pending requests. When a request occurs, and if the object state does not allow an immediate processing, the controller stores this request. This request is still stored until the controller allows the method execution.

3 Programming Controllers With BDL

In the previous examples, we have used operators to specify scheduling constraints concerning method executions. These operators allow to construct BDL expressions which specify scheduling constraints.

3.1 The operators of the BDL Language

In BDL, there are only two types: identifiers of methods and operators of scheduling. A BDL term corresponds either to a method identifier or to one (or two according to the arity of the operators) term and one operator. These operators are adapted from the asynchronous reactive language Electre [11]:

- an unary operator of *repetition*, quoted "`*`", indicates that the control specified by the term works indefinitely;
- a binary operator of *sequentiality*, quoted "`;`", indicates that the left term must be executed before the right one;
- two binary operators of *parallelism* indicates that the two terms (left and right) can be executed at same time. In that case, we consider two types of parallelism:
  - a parallelism so called *weak*, quoted "`|||`", expressing a *possibility*: the concurrent structure may be ended when a term has achieved its execution and the other has not started yet;
  - a parallelism so called *strong*, quoted "`||`", expressing a *necessity*: the concurrent structure is ended only when both terms have achieved their execution.
- a binary operator of *mutual exclusion*, quoted "`|`", indicates that executions of both terms are mutually exclusive;
- a binary operator of *priority*, quoted "`#`", indicates that if an execution request occurs for one of the methods in the right term, then the execution requests for methods in the left term are no longer satisfied (and they will be stored). However, the execution of methods in the right term is subordinated to the termination of all the methods being executed in the left term. In this case when no method of the left term is being executed and a request for a method of the right term occurs, then this one is immediately satisfied;
- two binary operators of *preemption* indicating that the execution of the left term can be stopped by the beginning of the execution of the right term. We consider both the following preemption types:
  - a preemption so called *weak*, quoted "`^`": the execution of the preemptive structure (right term) can take place only during the execution of the preempted structure (left term);

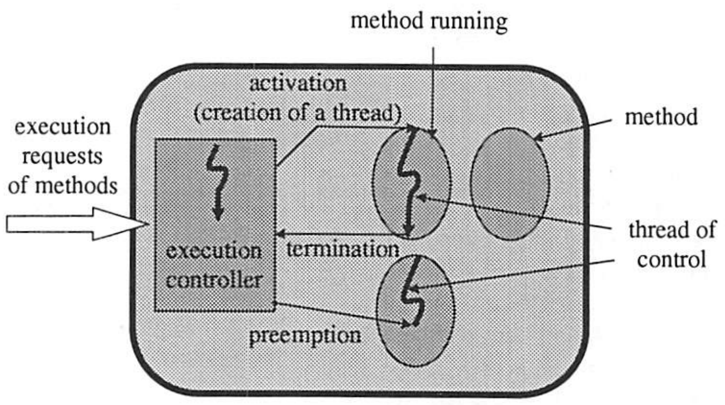


Figure 1: Architecture of a concurrent object with a execution controller

- a preemption so called *strong*, quoted "/": the execution of the preemptive structure is required even if the preempted structure has terminated its execution.

- all these operators have the same precedence level, so parenthesis could be used to modify this precedence.

Among these operators, the more complex is certainly the priority operator. We will illustrate its semantics by considering the `File` object (cf. 2.1) again. The previous specification was:

```
(open ; (write | read)* ; close)*
```

This specification introduces an unwanted behavior due to the semantic of the "*" operator which corresponds to an endless execution. Therefore `close` will never be executed. The use of preemption is unappropriate because it is too "rude". We wish to leave the reading (or writing) running until the end before closing the file by using a priority operator which allows to specify this type of control:

```
(open ; (write | read)* # close)*
```

Now, when receiving an execution request for `close`, then it is executed immediately if `read` (or `write`) is not running. Otherwise the requests for the two methods are not satisfied any longer and `close` starts as soon as one of the two methods terminates.

BDL operators enables one to easily describe different policies of use. A fair policy between both modes:

```
(open ; (write | read)* # close)*
```

or then give priority to reading:

```
(open ; (write # read)* # close)*
```

or writing:

```
(open ; (read # write)* # close)*
```

3.2 Examples of BDL programs

We give two examples of BDL programs with an object and a group of objects.

In a first example we illustrate the need of preemption using an elevator truck. This object has five methods: `init`, `m_on`, `m_back`, `m_up`, `m_down` and `stop`. The `init` method describes the initial position of the truck and must be executed before the four others. Methods allowing the truck to move along a same axis (left/right and up/down) must have mutually exclusive executions but the movement may be simultaneous along both of them. At last, if necessary, the truck can stop any movement by invoking the `stop` method. The mutual exclusion (operator "|") of the movement along a same axis is expressed by:

`m_on | m_back` and `m_up | m_down`

Possible parallel execution (operator "|||") is expressed by:

```
(m_on | m_back) ||| (m_up | m_down)
```

The preemption (operator "/") carried out by the `stop` method by:

```
((m_on | m_back) ||| (m_up | m_down)) /
                stop
```

Finally the global specification is:

```
                 init ;
(((m_on | m_back) ||| (m_up | m_down))
                / stop
```

In the second example, the control of the behavior of a group of objects (or *aggregate*) is achieved by using the behavior control for each object as a pattern. To illustrate how the behavior control of an aggregate can be defined, we use a simple video player [21] with four functions: load a tape, play a tape, stop playing and eject a tape. The video (`VideoPlayer` object) has two components: a motor (`Motor` object) and an eject mechanism (`EjectMech` object). The `Motor` object has two methods: `play` and `stop`. The `EjectMech` object has two methods too: `load` and `eject`.

The behavior control of the `Motor` object can be expressed by the following BDL code where the operator ^ means the execution of `play` method could be stopped by the execution of `stop` method:

```
(play ^ stop)*
```

and the one of the `EjectMech` by:

```
(load # eject)*
```

When we aggregate the two objects to build the `VideoPlayer` object, the behavior control of the `VideoPlayer` can be expressed by:

```
(load ; (play ^ stop)* # eject)*
```

One can notice in this code that hiding method identifiers of one of the two object produces the code of control of the other. This sort of aggregation (called *aggregation with hiding*) is conformed to the principle defined by Hartmann *et al.* [18]. This principle states that the behavior of an aggregate restricted to the methods of a component object must give the behavior of this component.

In the next section, we describe the choices that have been done for the implementation of BDL.

4 Semantics of BDL

4.1 An Event-Based Semantics

The notion of event is well suited to highlight the different steps in the processing of execution requests. These requests are caught by the controller as *arrival* events. These events represent requests either from other objects or from the object itself. It must be quoted that, in our model, these events occur at distinct instants and are separately received (one by one) by the controller. This distinction allows to model distributed objects more easily. The executions triggered by the controller can be considered as reactions to these *arrival* events. The execution is triggered by the emission of a *start* event towards the runtime system. A *termination* event informs the controller of the end of execution.

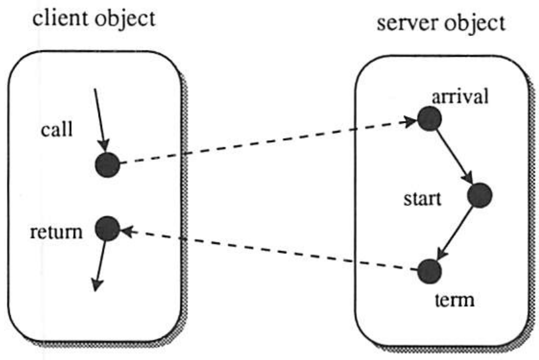


Figure 2: The events in the lifespan of a method invocation

When a method is called, what happens on the client object side must be also considered. The execution request is modeled by a *call* event and once the method is correctly carried out, a *return* event is sent back to the calling object.

Figure 2 describes an event sequence happening when an object (client) requests the execution of a method from another object (server).

About the called object, the distinction between *arrival* and *start* events is important because it stands for the part played by the controller upon the execution of a method. The distinction between *start* and *term* events introduces the notion of duration for an execution. This model of execution is also present in [19] under the name SOS (Service Object Synchronisation).

4.2 Using an Automaton as Target Code

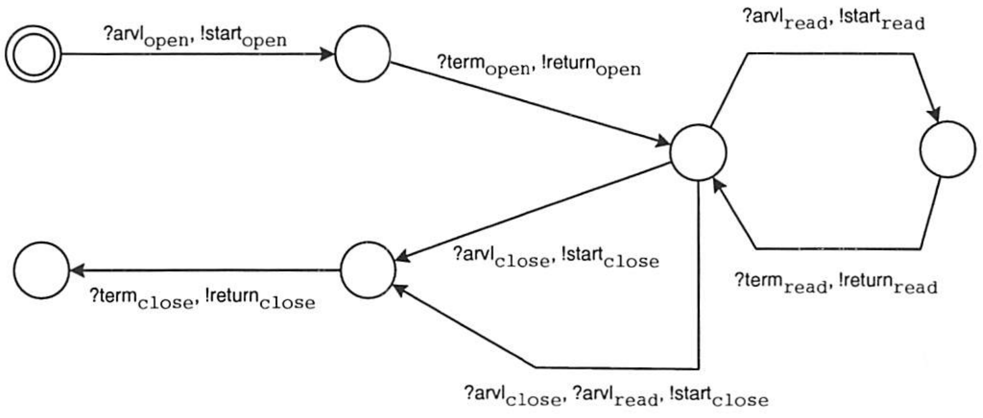
From a theoretical point of view, a BDL program ensures a correct event trace. An automaton is a well-known structure to achieve this work. For example, the BDL program

```
(open ; read* # close)
```

could be represented by the automaton of figure 3.

Using an automaton as target code for BDL presents the following interests:

- *efficiency*: an automaton described in a programming language produces a fast executable code. This efficiency is important because this code is often executed and the object must quickly respond to a execution request;
- *proofs*: the automata are mathematical structures on which many verification tools have been developed.



?arvl$_{open}$, !start$_{open}$
?term$_{open}$, !return$_{open}$
?arvl$_{read}$, !start$_{read}$
?term$_{read}$, !return$_{read}$
?arvl$_{close}$, !start$_{close}$
?term$_{close}$, !return$_{close}$
?arvl$_{close}$, ?arvl$_{read}$, !start$_{close}$

Figure 3: The (simplified) automaton of the BDL program `(open ; read* # close)`

4.3 From BDL Program to an Automaton

To produce an automaton from a BDL program is possible but a great part of this work is already done (and well done because it concerns critical systems) by a family of languages named *reactive languages*.

Reactive languages have been designed to program reactive systems. A reactive system [16] is defined as reacting instantaneously to events received continually from the environment by emitting events towards it. The system does not compute or carry out a function but maintains a balance with its environment. That is: maintains a relationship between its inputs and outputs as time goes past. Most of real-time systems such as control or signal processing systems and communication protocols are reactive. Software implementation of these systems has led to develop a family of languages so called reactive among which it may be quoted Esterel [3], Statecharts [17] and Electre [11]. Due to the critical aspect of the systems implemented, these languages own a mathematical semantics allowing formal verification of properties on the behavior of these systems. The compilers of reactive languages produce an automaton used both by verification tools to verify properties and by translators to generate a C (or Ada) code describing the automaton.

In a previous section (see 4.1), we have shown the method execution could be represented by a sequence of events, the controller managing this sequence. Thus the controller behaves as a reactive system.

4.4 The Esterel Language

We choose to use the synchronous reactive language Esterel [3] for it offers high level control structures and, on the other hand, it is interfaced with different verification tools. Though its execution mode is synchronous (simultaneous perception of several events) we use it with asynchronous way to describe the working of the execution controller. The perception of events is restricted to only one event per instant.

Esterel is a synchronous imperative language. A quick introduction to Esterel semantics can be found in [4]. A program in Esterel consists of a collection of interacting modules. A module has an interface that defines its input and output signals and a body that is an executable statement.

There are two basic composition operators: the parallel composition operator "| |" and the sequential composition operator ";". In a parallel statement, all components are activated simultaneously; the parallel statement terminates instantaneously when both components have terminated.

Interactions between modules takes place through the use of "*signals*". A signal may carry a value. Occurrences of signals that are emitted by a program's environment (input signals) are the unique causes for the program to react. Input signals correspond to input events from the model of controller. An Esterel program reacts instantaneously to the receipt of input signals by emitting output signals towards its environment. Output signals correspond to the activation events from our model. A program may also emit and receive internal signals, used for inter-module communication within the program itself. Internal signals are not visible from the environment.

Two assumptions are made on signals:

- signals are broadcast within the program (ie. each module in the program receives all signals);
- signals are received instantaneously by all modules in the program.

An Esterel program does not have an associated clock.

```
trap T_EXIT in
   signal TERMINATED in
      trad(T1);
      emit TERMINATED
   ||
      abort
         await immediate TERMINATED do
            exit T_EXIT
         end                    V
      when immediate [                 STARTm ]
   end signal              m ∈ RTS(T2)
||
   signal TERMINATED in
      trad(T2);
      emit TERMINATED
   ||
      abort
         await immediate TERMINATED do
            exit T_EXIT
         end                    V
      when immediate [                 STARTm ]
   end signal              m ∈ RTS(T1)
end trap
```

Figure 4: Semantics of BDL expression "T_1 | | | T_2" expressed with Esterel operators

The synchrony hypothesis (reaction takes no time) coupled with signal assumptions allows a rigorous processing of multiform time. Time can be handled as an ordinary signal ("clock signals"); any signal defines a particular clock.

4.5 Defining BDL Semantics from Esterel Operators

Some BDL operators have a direct equivalent Esterel operator as "*", ";", "| |" operators. But the definition of "| | |", "|", "#", "/", "^" operators needs a sequence of Esterel statements. The definition of the whole operators is too long for this paper, the reader will refer to [5] for the complete semantics.

We present here, as an example, the translation of the weak parallelism operator ("| | |") depicted by figure 4. Because of the semantics of "| | |" operator, when a branch terminates whereas the other branch has not yet started, an expression such as T_1 | | | T_2 terminates (each term T_i may be composed recursively by other BDL operators).

So, in the Esterel code, at end of each branch *trad(T_i)*, a `TERMINATED` event (local to each `signal` statement) is emitted. The `TERMINATED` event throws the `T_EXIT` exception only if a `START` event (belonging to *RTS* set) concerning a method of the other branch has not yet been emitted. This is done by Esterel preemption operator (`abort` ... `when`). We will explain the meaning of *RTS* (*Ready To Start*) set. The *RTS* set is an attribute calculated on each term T giving the name of the methods ready to be executed that are located on the left part of BDL sequential expression. For example

$$RTS(\texttt{((A ; C) | (B ; D))}) = \{\texttt{A}, \texttt{B}\}$$

Disjunction between the different `START_m` events corresponds to different possibilities of executions of m methods.

4.6 Esterel Architecture of an Execution Controller

An overview of Esterel architecture for an execution controller managing the execution of two methods (`A` and `B`) is represented on figure 5.

Each method managed by the controller needs three modules:

- a `METHOD_STATE` module has in charge two functions. The first one indicates the execution state at every instant by emitting (or not) the `ACTIVE` event. The second one to emit a `CHANGE` event towards `BUFFER` module at every method termination to request them to emit their pending requests again. Therefore a pending request can be executed only when an execution terminates.
- a `BUFFER` module that stores pending requests then emits them again (`REQUEST` emission) when receiving a `CHANGE` event. When a method is allowed to run, the module is informed by the reception of a `START` event.
- a `REQUEST_HANDLER` module receiving requests (`ARRIVAL` event corresponding to the *arrival* event on figure 2). The request is then transmitted (`REQUEST`) to `BEHAVIOR` module. If the request cannot be served then it is sent to the `BUFFER` module to be stored.

In the execution controller, on figure 5, there is only one `BEHAVIOR` module representing the decision structure. The `BEHAVIOR` module implements the BDL program using Esterel language. We have implemented a compiler carrying out the compilation of BDL program into Esterel code and also building the main module Esterel representing the execution controller.

The Esterel compiler allows to get a finite state automaton represented by an intermediate code (`oc`) that is translated in C by a postprocessor. Of course the trouble is the blow up of the size of automaton when the number of methods to control becomes important. Nevertheless, for a small number of methods, the size of the object

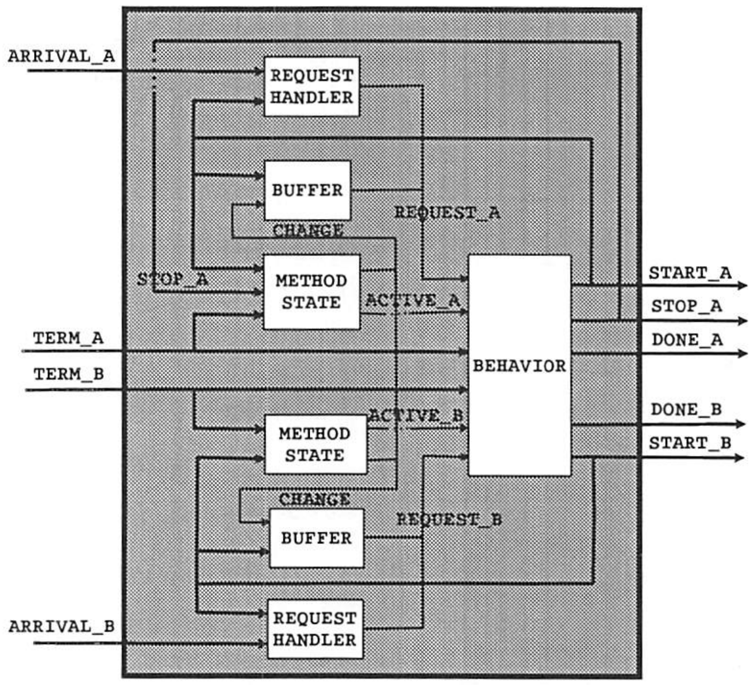


Figure 5: Esterel architecture of an execution controller managing two methods

code is not too big: for six methods managed in mutual exclusion, we have an object code of 5 Kbytes[1]. There is another alternative to reduce this size. One may use another mode of Esterel compiler that uses boolean circuits instead of automata but the verification tools working for this new data structure are under development.

4.7 Implementing a Concurrent Object with an Execution Controller

The implementation of concurrency in the reactive object model relies on the use of the *lightweight processes* library C Threads [13] built on the Mach operating system [25]. The creation of a reactive object implicates the creation of a process holding at least a lightweight process ensuring the object control by receiving the execution requests and by executing the execution controller code. Each method execution gives way to the creation of a lightweight process.

The object-oriented language used is C++. When creating an object, the constructor call involves the creation of a process bound to a communication port. The object is registered to a name server (functionality offered by Mach facilitating object distribution), and the lightweight process ensuring the object control is activated. These mechanisms are gathered in a `ReactiveObject` class which every reactive object must inherit.

[1] Object code obtained on a SparcStation 5 with `cc` of SunOS 4.1.4.

The implementation of reactive objects is divided into three steps. The first step consists in building the controller from a BDL program. The second one consists in the controller implementation which is obtained by linking the code of the controller to the code of the storage structure. The last one consists of linking the execution controller code with the code of the object and with the code ensuring thread management and communication between reactive objects. Figure 6 depicts these differents steps.

In the current version, every object has its own reactive script that is a program representing an automaton. That is trouble for the implementation of large applications where the number of reactive objects handled is significant. Nevertheless, in a new implementation, this problem will be solved by using only one reactive script for every object class. The reactive script then acts as a server which the instances submit their current state and the received event. In response, the automaton sends back the new state and the events to emit.

5 Verification of The Control Behavior

Verification is an important step in the lifespan of an object. Object-oriented programming is concerned with reusing: an object designed for an application may be reused in other applications. To avoid an error in the

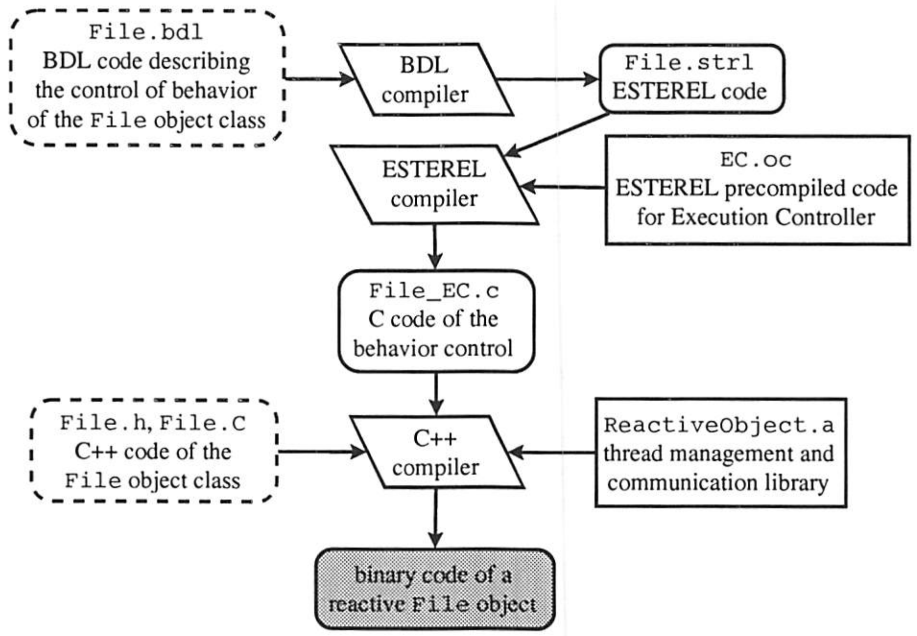

File.bdl
BDL code describing the control of behavior of the File object class
BDL compiler
File.strl
ESTEREL code
ESTEREL compiler
EC.oc
ESTEREL precompiled code for Execution Controller
File_EC.c
C code of the behavior control
File.h, File.C
C++ code of the File object class
C++ compiler
ReactiveObject.a
thread management and communication library
binary code of a reactive File object

Figure 6: Development steps of a reactive `File` object

design of an object being propagated to several applications, it is important to make sure of *correct working* of this object.

In respect to correct working, we mean that the object must be able to serve execution requests of methods permanently as well as respecting the control specification of its behavior. So we must ensure that a case (sequences of execution requests) leading to a "blocking" of the object does not exist. These blocking cases correspond to a problem frequently encountered within concurrent programming: the deadlock.

For example, let us take an object having two methods met_1 and met_2 and the behavior control of which is specified by the following expression

$$(met_1 \;|||\; met_2)*$$

On the other hand, let us consider the met_1 method calling met_2 during its execution.

If met_2 is the first method being executed, a deadlock may appear if, during the execution of met_2, the execution of met_1 starts. The execution of met_1 needs to execute a the call of met_2. As met_2 is being executed, met_1 must wait for the end of the current execution of met_2 to know if its request is satisfied. However, when met_2 is terminated, it is no longer ready for execution

$$(met_1 \;|||\; met_2)* =$$
$$(met_1 \;|||\; met_2) \;;\; (met_1 \;|||\; met_2)*$$

So the execution request for met_1 cannot be satisfied any longer and so met_1 cannot achieve its execution. That leads to the case when neither of these methods can be executed any longer.

Because of our approach this problem can be detected formally. If we are able to statically[2] determinate the call graph of the methods of an object, it is then possible to build Esterel modules expressing this information. These modules are compiled with the module representing the execution controller to generate an automaton on which it is possible to determinate deadlock states. These modules "simulate" the execution of a method and communication between methods may be introduced to allow verification.

The automaton, on which the verification is carried out, is obtained after compiling an Esterel simulation module composed of a controller module and modules representing the execution of every method managed by the controller. For this simulation module, the *start* and *term* events are considered as internal events (to the object) and so are not visible in the interface of simulation module. This interface takes as inputs the *request* events corresponding to the methods managed by the execution controller and the *done* events of called methods. The `done` events of the methods managed by the controller and the `request` events of the called methods are outputs.

This modular architecture allows a modular approach of verification by checking correctness at the object level. In assembling the simulation modules of several objects

[2]This constraint excludes obviously the mechanisms such as function pointers.

it is possible to check the correctness of the behavior of a group of objects.

Compiling a simulation module, we get an automaton on which we can verify properties with the Fc2Tools [7] verification tool developed in INRIA[3].

Fc2Tools is designed as the set of programs using the same automaton format (fc2). The internal representation of automata is carried out using binary decision diagrams (BDD) and so a more efficient representation in memory is allowed. So the limits on the size of automata to be verified are extended. If a deadlock is detected, then an event sequence leading to this deadlock is generated by the tool.

A small example is that of a micro-wave oven. This object is compound of two objects: a `Gate` and a `MWGenerator` (micro-wave generator). The behavior control of the `Gate` object is the following:

```
(open ; close)*
```

and that of the `MWGenerator` is:

```
(on ; off)*
```

The behavior control of the aggregate can be defined as:

$$(\boxed{\texttt{(open ; close)}}\ \#^{4}\ \boxed{\texttt{(on ; off)}}\ )*$$

But we can refine this control by precising the user to open the door while the generator is working. As this is a dangerous situation, the emission of micro-waves must be stopped:

```
          ((open ; close)
   # (on ^ (open ; close) ; off))*
```

From the automaton generated by this specification, it is possible to verify the case where the generator works and the gate is open never occurs.

6 Related Work

Many works have been led to the area of the control of concurrency in the concurrent object-oriented languages. Nevertheless as far as we know few works have used a reactive model to describe the execution control in an object.

Concerning the control of concurrency in the concurrent object-oriented languages, we can distinguish four approaches:

[3]Institut National de Recherche en Informatique et Automatique – France.

[4]The priority operator "#" expressed here a notion of possibility, the working of the generator being not mandatory before open it again.

- the mechanisms based on the *synchronization counters* developed by ROBERT and VERJUS [24] in which the execution state of methods is determinated by updating a set of three counters *arrival*, *start*, and *term*. These counters are mainly used in guards: conditions associated to the activation of a method. We find these counters again in different languages such as Guide [15] or Dragoon [14]. If the approach has a great power of expression, using it is still difficult owing to its complexity.

- the *path expressions* [9] use a notation derived from regular expressions to specify synchronization constraints. We find these expressions in Procol [28]. These expressions have influenced the designers of Electre too and thus indirectly the BDL operators.

- in Eiffel// [10] there exists a special method ensuring the request processing. This method presents many similitudes with the execution controller. However our approach owns a management of the intra-object concurrency and is supported by a formal model.

- in the *enabled-sets* which has been developped with the Rosette language [27], the control is defined by defining *states of control*. For these states, a specific set of methods allowed to be executed is defined.

Concerning the works using reactive language and object, there are two main ways:

- the Objectcharts [12] which use the Statecharts. The Statecharts represent a reactive language based on a visual formalism. If this formalism is pleasant to use, the designers are not very clear about the executability of the specifications and on its implementation.

- BOUSSINOT's works [8] that consist of defining a prototype of an object-oriented language based on a synchronous execution model. In this approach, a reactive object holds attributes and methods. These last ones are "reactive agents". Opposite to our approach, these methods are reactive code and the aimed purpose is to structure a reactive program with an object-oriented approach.

7 Conclusion And Future Works

We propose BDL, a language to control the behavior of concurrent objects. This language allows to program a special entity in the object: the execution controller. Adding an execution control allows a concurrent object to make it permanently receptive to its environment. This feature of reactive objects is important for the reliability of an application because it guarantees an answer to the object requesting a service. The nature of the answer depends upon the behavioral logic of the receiver object and its state. The reactive object allows to preempt executions that allows its adaptation to the modification of its environment.

Using reactive objects is twofold: it allows intra-object concurrency management and fast adaptation to external stimuli. A reactive object offers the possibility of execution concurrently several methods with respect to a clearly expressed semantics.

This model extends the object reusability in two ways. The definition of an execution controller as a complete entity offers the possibility of modifying the behavior of an object in a quite simple manner by replacing the existing controller with a new one. This replacing must be of course made respecting the type of its inputs and outputs.

Using an automaton as target code enables us to dispose of a mathematical model upon a certain number of properties can be verified. Proofs enable us to control that the initial specification have been correctly translated.

BDL cannot express conditions of activation related to objet attributes. This restriction is a constraint imposed by verification tools. We are now working towards this way be using Toupie [23] a constraint language working on finite domains instead of Esterel.

8 Acknowledgments

The first author is supported by a grant 94/RPC-B-94-85 from Conseil Régional Poitou-Charentes.

The authors are grateful to G. LOUIS for their comments and suggestions.

References

[1] P. America. POOL-T: A parallel object-oriented language. In B.D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 199–220. MIT Press, Cambridge, Massachusetts, 1987.

[2] M. Augeraud. A reactive part to specify dynamic objects behavior. Indo-French workshop on object-oriented systems, November 1992.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992. Postscript version available[5].

[4] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating Reactive Processes. In *Proceedings of the 20th ACM Conference on Principles of Programming Languages*, pages 85–98, Charleston, South Carolina (USA), January 1993. Postscript version available[6].

[5] F. Bertrand. *Un modèle de contrôle réactif pour les langages à objets concurrents*. PhD thesis, Université de La Rochelle, L3I, Avenue Marillac, 17042 La Rochelle Cedex, January 1996.

[6] F. Bertrand and M. Augeraud. Control of object behavior : asynchronous reactive objects. In *Proceedings of International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, pages 539–544, Hong Kong, May 1994. Chinese University of Hong Kong. Postscript version available[7].

[7] A. Bouali, A. Ressouche, R. de Simone, and V. Roy. *The* FCTOOLS *Reference Manual*. INRIA / ENSMP-CMA, Sophia-Antipolis, FRANCE, 1994. Postscript version available[8].

[8] F. Boussinot, G. Doumenc, and J.B. Stephani. Reactive Objects. Technical Report RR-2664, INRIA, 2004, Route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex, FRANCE, October 1995. Postscript version available[9].

[9] Roy H. Campbell and N. Haberman. *The Specification of Process Synchronization by Path Expressions*, pages 89–102. Springer Verlag, December 1973.

[10] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[11] F. Cassez and O. Roux. Compilation of the ELECTRE reactive language into finite transition systems. *Theoretical Computer Science*, 146, July 1995.

[12] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or how to use Statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, 1992.

[13] E. C. Cooper and R. P. Draves. C THREADS. Department of Computer Science, Carnegie Mellon University, September 1990.

[5] ftp://ftp-sop.inria.fr/meije/esterel/papers/BerryGonthierSCP.ps.gz

[6] ftp://ftp-sop.inria.fr/meije/esterel/papers/popl.ps.gz

[7] http://www.univ-lr.fr/Labo/L3I/L3I/equipe/fbertran/papers.html

[8] ftp://ftp-sop.inria.fr/meije/verif/primer.ps.gz

[9] ftp://zenon.inria.fr/pub/rapports/RR-2664.ps

[14] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini. Definition of reusable concurrent software components. In *Proceedings of ECOOP'91*, pages 148–166, Geneva, SWITZERLAND, July 1991. Springer Verlag.

[15] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveil, and X. Rousset de Pina. A Synchronization Mechanism for Typed Objects in a Distributed System. *ACM SIGPLAN Notices*, 24(4), april 1989.

[16] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[17] D. Harel. STATECHARTS: a visual formalism for complex systems. *Science Of Computer Programming*, 8:231–274, June 1987.

[18] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 57–77. Springer Verlag, 1992.

[19] C. Mac Hale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* PhD thesis, University of Dublin, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1994. Postscript version available[10].

[20] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[21] A. M. D. Moreira. *Rigorous Object-Oriented Analysis.* PhD thesis, University of Stirling, Department of Computing Science and Mathematics, Stirling FK9 4LA, August 1994. Postscript version available[11].

[22] M. Papathomas. *Language design rationale and semantic framework for concurrent object-oriented programming.* PhD thesis, Université de Genève, January 1992. Postscript version available[12].

[23] A. Rauzy. *Toupie v. 0.26 User's Manual.* LaBRI (URA 1304), 351, Cours de la Libération, 33405 Talence Cedex, FRANCE, October 1994. Postscript version available[13].

[10]ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-86.ps.gz

[11]ftp://ftp.cs.stir.ac.uk/pub/tr/cs/1994/TR132.ps.Z

[12]ftp://cui.unige.ch/OO-articles/papathomasThesis.ps.Z

[13]ftp://ftp.u-bordeaux.fr/pub/Local/Info/Publications/Rapports-internes/RR-58693.ps.Z

[24] P. Robert and J.-P. Verjus. Toward autonomous descriptions of synchronization modules. In B. Gilchrist, editor, *Information Processing 77*, pages 981–986. North Holland Publishing Company, 1977.

[25] E. Sheinbrood. *The design of the* MACH *Operating System.* Prentice Hall, February 1993.

[26] M. Tokoro. The society of objects. In *Addendum to the Proceedings of OOPSLA'93*, pages 3–11, Washington, D. C., October 1993. ACM. Invited address.

[27] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of OOPSLA'89*, pages 103–112, New Orleans, Louisiana, October 1989.

[28] J. van den Bos and C. Laffra. PROCOL: A concurrent object-oriented language with protocols delegation and constraints. *Acta Informatica*, 28(6):511–538, 1991.

A Domain-Specific Language for Regular Sets of Strings and Trees

Nils Klarlund

AT&T Labs–Research
klarlund@research.att.com

Michael I. Schwartzbach

BRICS, University of Aarhus
mis@brics.dk

Abstract

We propose a new high-level programming notation, called FIDO, that we have designed to concisely express regular sets of strings or trees. In particular, it can be viewed as a domain-specific language for the expression of finite-state automata on large alphabets (of sometimes astronomical size).

FIDO is based on a combination of mathematical logic and programming language concepts. This combination shares no similarities with usual logic programming languages. FIDO compiles into finite-state string or tree automata, so there is no concept of run-time. It has already been applied to a variety of problems of considerable complexity and practical interest.

In the present paper, we motivate the need for a language like FIDO, and discuss our design and its implementation.

We show how recursive data types, unification, implicit coercions, and subtyping can be merged with a variation of predicate logic, called the Monadic Second-order Logic *(M2L) on trees. FIDO is translated first into pure M2L via suitable encodings, and finally into finite-state automata through the MONA tool.*

1 Introduction

Finite-state problems are everywhere, embedded in many layers of software systems, but are often difficult to extract and solve computationally. This basic observation is the motivation for the work presented in this paper.

Recent research by us and our colleagues has exploited the Monadic Second-Order Logic (M2L) on finite strings and trees to solve interesting and challenging problems. In each case, the results are obtained by identifying an inherent regularity in the problem domain, thus reducing the problem to questions of regular string or tree languages. Successful applications today include verification of concurrent systems [9, 8], hardware verification [2], software engineering [10], and pointer verification [7]. Work in progress involves a graphical user interface for regular expressions extended with M2L and document logics for the WWW.

The rôle of M2L in this approach is to provide an extraordinarily succinct notation for complicated regular sets. Our applications have demonstrated that this notation in essence can be used to describe properties, where finite state automata, regular expressions, and grammars would be tend to be cumbersome, voluminous, or removed from the user's intuition. This is hardly surprising, since M2L is a variation on predicate logic and thus natural to use. Also, it is known to be non-elementarily more succinct than the other notations mentioned above. Thus, some formulas in M2L describe regular sets for which the size of a corresponding DFA compared to the size of the formula is not bounded by any finite stack of exponentials.

The flip side of this impressive succinctness is that M2L correspondingly has a non-elementary lower bound on its decision procedure. Surprisingly, the MONA implementation of M2L [5] can handle non-trivial formulas, some as large as 500,000 characters. This is due in part to the application of BDD techniques [4], specialized algorithms on finite-state automata [3], and careful tuning of the implementation [11]. Also, it turns out that the intermediate automata generated, even those resulting from subset constructions, are usually not big compared to the automata representing the properties reasoned about.

The successful applications of M2L and MONA reside in a common, productive niche: they require the specification of regular sets that are too complicated to describe by other means, but not so complicated as to be infeasible for our tools.

While the basic M2L formalism is simple and quite intuitive, early experience quickly indicated that this formalism in practice suffers from its primitive domain of discourse: bit-labeled strings and trees. In fact, M2L specifications are uncomfortably similar to assembly code programs in their focus on explicit manipulations of bit patterns. For M2L interpreted on trees, the situation is even worse, since the theory of two or more successors is far less familiar and intuitive than the linear sublogic.

Similarly to the early experiences with machine languages, we found that M2L "programmers" spent most of their time debugging cumbersome encodings.

Our contributions

In this paper, we propose a domain-specific programming formalism FIDO that combines mathematical logic and recursive data types in what we believe are new ways.

We suggest the following four kinds of values: finite domains, recursive data values (labeled by finite domains), positions in recursive data values, and subsets of such positions. We show that many common programming language concepts (like subtyping, coercions, and unification) make sense when the underlying semantics is based on assigning an automaton (and not a store transformer) to expressions.

This semantic property allows us to view the compilation process as calculations on values that are deterministic, finite-state automata, just as an expression evaluator calculates on numbers to arrive at a result. That is, automata are the primitive objects that are subjected to operations reflecting the semantics of the language.

This view is quite different from the method behind most state-machine formalisms used in verification (such as the Promela language [6]): a language resembling a general purpose language expresses a single finite-state machine, whose state space and transition system is constructed piecemeal from calculations that explore the state space.

Our view, however, is similar to some uses of regular expressions for text matching, except that most implemented algorithms avoid the construction of deterministic automata.

FIDO is implemented and provides, along with supporting tools, an optimizing compiler into M2L formulas. It has been used for several real-life applications and is also the source of the biggest formulas yet handled by MONA.

In this article, we motivate and explain FIDO. In particular, we discuss the type system and compilation techniques. We also give several examples (some taken from articles already published, where we have used FIDO without explaining its origin or design). Some technical considerations concerning the relationship between our data structures for tree automaton representation [3] and the compilation process will be explained elsewhere.

2 M2L and MONA

Basic M2L has a very simple syntax and semantics. Formulas are interpreted on a binary tree (or a string) labeled with bit patterns determining the values of free variables. First-order terms (t) denote positions in the tree and include first-order variables (p) and successors ($t.0$ and $t.1$). Second-order terms (T) denote sets of positions (i.e. *monadic* predicates) and include second-order variables (P), the empty set ($\emptyset$), unions ($T_1 \cup T_2$), and intersections ($T_1 \cap T_2$). The basic predicates are set membership ($t \in T$), equality ($t_1 = t_2$), ancestor relation ($t_1 < t_2$), and set inclusion $T_1 \subseteq T_2$). The logic permits the usual connectives ($\wedge$, $\vee$, $\neg$) and first and second-order quantifiers ($\forall_1$, $\exists_1$, $\forall_2$, $\exists_2$). By convention, a leaf is a position p for which $p = p.0$ and $p = p.1$. The sublogic for strings uses only the 0-successor.

The MONA tool accepts such formulas in a suitable ASCII syntax and produces a minimum DFA that accepts all trees satisfying the given formula. Thus, satisfiability of a formula is equivalent to nonemptyness of the derived automaton, and validity is equivalent to totality. The values of free variables in the formula are encoded in the alphabet of the automaton. Thus, a formula with 32 free variables yields an alphabet Σ of size 2^{32}. In the internal representation of these automata, the transition function is shared, multi-terminal Σ-BDD. With these BDD techniques, the MONA tool has processed formulas with hundreds of thousands of characters in a few minutes.

3 The Motivation

A small example will motivate the need for a high-level notation. Assume that we wish to use MONA to prove the following (not too hard) theorem: for every string in $(a+b)^*c$, any a is eventually followed by c.

To state this theorem in M2L, we must first choose an encoding of the labels a, b, and c. For this purpose we introduce two free second-order variables X_0 and X_1. The labels can be encoded according to the following (arbitrary) schema: a position p has label a if $p \notin X_0 \wedge p \notin X_1$, that is, a corresponds to the bit pattern 00. Similarly, we can assign to b the bit pattern 01 and to c the pattern 10. The property "a is eventually followed by c" becomes the formula:

$$\begin{aligned}\psi \equiv \forall_1\, p : (p \notin X_0 \wedge p \notin X_1) \Rightarrow \\ (\exists_1 q : p < q \wedge (q \in X_0 \wedge q \notin X_1))\end{aligned}$$

The regular expression $(a+b)^*c$ can in a similar way be encoded as the formula:

$$\begin{aligned}\phi \equiv \forall_1\, p : (\neg(p \in X_0 \wedge p \in X_1)) \wedge \\ ((p \in X_0 \wedge p \notin X_1) \Leftrightarrow p = p.0)\end{aligned}$$

and the theorem above is then formally stated as the implication $\phi \Rightarrow \psi$. The MONA tool will readily verify that this formula is an M2L tautology, thus proving our theorem.

A reason for M2L specifications being much more voluminous than promised should now be apparent: there is a significant overhead in encodings. Moreover, there are no automatic checks of the consistent use of bit patterns.

Support for such encodings is usually supplied by a type system. For M2L on strings, regular sets immediately suggest themselves as notions of types. It is quite common for M2L formulas to be of the implicational form $\phi \Rightarrow \psi$, where ϕ is a formula restricting the strings to a coarse regular set and ψ provides the more intricate restrictions. Thus, a high-level version of the above formula could look like:

```
string x: (a+b)*c;
∀pos p:x.(p=a ⇒ ∃pos q:x.(p<q ∧ q=c))
```

The keywords **string** and **pos** are intended to declare free variables of these two kinds. This formula can be read as: "for all positions p in the string x, if p has label a, then there exists a position q, also in x, such that p is before q and q has label c". The main formula is almost the same as the MONA version, but the proper use of labels is now supported by the compiler and can be verified by a type checker.

For M2L interpreted on trees, however, there is no intuitive analogue to regular expressions. But from programming languages we know an intuitive and successful formalism for specifying coarse regular sets of trees: recursive data types. Thus, we adopt a well-known and trusted programming concept into our high-level notation. Using this idea, we may prove our theorem as follows:

```
type T = a,b(next: T) | c;
string x: T;
∀pos p:x.(p=a ⇒ ∃pos q:x.(p<q ∧ q=c))
```

Arbitrary recursive data types may of course be expressed directly as formulas, but the translation is voluminous and best performed automatically. The translation also solves the problem that the Mona decision procedure works on formulas whose domain of discourse is only binary trees, whereas values of recursive data types are trees with a varying number of branches. (The solution is rather technical, since it involves bending the recursive data type value into the shape of a binary branching tree.)

Note that not all regular tree sets can be captured by recursive data types. Consider binary trees, in which nodes are colored red, green, or blue. The subset of trees in which at most one node is colored blue is *not* a recursive data type; however, it is easily captured by the following FIDO specification:

```
type RGB = red,green,blue(left, right: RGB) | leaf;
tree x: RGB;
∀pos p,q: x.(p=blue ∧ q=blue ⇒ p=q)
```

Certainly, more advanced and complicated notions of data types could similarly be adopted [1]. However, the FIDO philosophy is to rely heavily on *standard* programming language concepts to describe complex structures and operations. The ambition is that these idioms should be merged seamlessly with logical concepts that describe complex properties of such structures.

In general, we allow finite domains (from which the name FIDO derives) to be the values of nodes. Finite domains are constructed conjunctively and disjunctively from enumerated and scalar types. Thus the alphabets of tree automata reading such recursive data types easily become very large.

4 The Design

While this paper is not intended as a proper language report, we will explain the more interesting or unusual concepts that the FIDO notation provides.

Domains and Data Types

Finite domains are constructed from simple scalar lists, freely combined with a product operator (&) and a union operator (|). When the union of two finite domains is formed, it is required that they are disjoint. Thus, if we define the domains:

```
type Turn = [1..2];
type PC = a,b,c,d;
type State = PC & PC & Turn;
```

then a value of the domain State may be written as State:[a,b,2]. From the more complicated definitions:

```
type A = a1,a2;
type B = b1,b2,b3;
type C = A | B;
type D = A & B & C;
type E = C & D;
```

we obtain values as: E:[a1,[a2,b3,[a2,b1]]]. In formulas, finite domain values may be unified using a syntax such as State:[pc?,a,r?], where ps and r are unification variables.

The recursive data types are quite ordinary, except that the constructors are generalized from single names to finite domains.

The finite domains could of course be encoded as (non-recursive) data types. We have chosen to have a separate concept for several reasons. First, the distinction between trees and their labels seems intuitive for many applications. Second, we can allow more operations on finite domains that on trees; for example, the introduction of unification or concatenation on trees would yield an undecidable formalism. Third, in the translation into automata, finite domains are encoded in BDDs whereas trees are encoded in the state space; often, it is necessary for the programmer to control this choice. An example is:

```
type Comp = State(next: Comp) | done;
```

which is a linear data type of sequences of state values terminated by a node labeled done. A non-linear example is:

```
type Tree = red,black(val: Enum,
                      left,right: Tree) |
            leaf;
type Enum = [1..10];
```

denoting some binary trees. The notation [1..10] abbreviates the corresponding 10 scalars.

Variables

There are four kinds of variables in FIDO. We introduce them by examples. A *domain variable* s that ranges over states may be declared as

```
dom S: State;
```

Tree variables (recursive data type variables) x and y may be declared as:

```
tree x,y: Tree;
```

Each variable defines its own space of positions. Thus, a position in x cannot be used to denote a node in y. To declare a *position variable* that may denote positions in either x or y, we write:

```
pos p: x, y;
```

A value of this variable points to a node in either x or y, but in any case, the node pointed to is either red or black. Similarly, a *set variable* S containing positions in the union of x's and y's position spaces may be declared as:

```
set S: x, y;
```

Quantification

All variables can be quantified over. For example, the formula "there is a computation that contains a loop" may involve quantification over both strings (trees), finite domains, and positions:

```
∃string x: Comp. ∃dom s: State.
    ∃pos p,q: x.(p<q ∧ p=s ∧ q=s)
```

Types

A type may have one of four different kinds: pos, set, dom, and tree. The pos kind corresponds to first-order terms, i.e. positions in trees; the set kind similarly encompasses second-order terms; the dom kind is new compared to M2L and describes values of finite domains; finally, the tree kind is a further extension that captures entire trees as values.

Within each kind, a type is further refined by a set of tree names and a set of data type names. For example, the type (pos,{x,y},{R,S,T}) denotes positions of nodes in either the tree x or y that are roots of subtrees of one of the data types R, S, or T. These refined types prove to be very convenient in restricting free variables in the model and in expressing relativized quantifications. Furthermore, this type structure proves crucial for optimizations in the implementation.

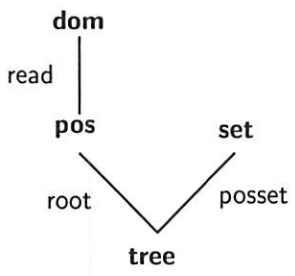
The type rules impose restrictions on all operators in the language. Generally, the rules boil down to trivial statements about finite sets. For example, if the terms s_i have types (set,X_i,D_i), then $s_1 \cap s_2$ has type (set,$X_1 \cap X_2$,$D_1 \cap D_2$). Also, if the term p has type (pos,X,D), then the term p.n has type (pos,X,{T.n | T∈ D}), where T.n is the data type reached from T along an n-successor.

Some formulas can be decided purely on the basis of the type system. For example, if p has type (pos,X_p,D_p) and s has type (set,X_S,D_S), then the formula $p \in S$ is false if $X_p \cap X_S = \emptyset$ or $D_p \cap D_S = \emptyset$. Such static decisions are exploited by the FIDO compiler.

Notational Conveniences

A formal notation has a tendency to become a quagmire of details. In the design of FIDO, we have attacked this problem in three different ways.

First, it is often convenient implicitly to *coerce* values between different kinds. This we have expressed through a simple subtype structure. Two types (κ_1,X_1,D_1) and (κ_2,X_2,D_2) are related by the subtype order if $X_1 \subseteq X_2$, $D_1 \subseteq D_2$, and κ_1 is below κ_2 in the following finite order:



The order relations have been decorated with coercions functions: posset computes the set of positions in a tree, root finds the root positions of a tree, and read computes the label of a position. This subtype structure is exploited to automatically insert coercions. Note that our subtype structure clearly is semantically coherent, so that coercions are unique [12]. If we added the coercion: singleton: **pos** → **set**, then semantic coherence would fail.

Second, we allow implicit *casts* between finite domains. For example, in the definitions:

```
type Fruit = apple,orange;
type Root = carrot,potato;
type Vegetable = Fruit | Root;
```

we will allow values of the domains Fruit and Root to be used directly as values of the domain Vegetable, even though they strictly speaking should be expressed as e.g. cast(Fruit:apple,Vegetable).

Third, we allow sensible *defaults* whenever possible. Thus, if a name can unambiguously be determined to have a specific meaning, then all formal qualifiers may be dismissed. For example, if the name orange is only used as a scalar in the domain Fruit, then the constant Fruit:orange may be written simply as orange.

As a specific example of these techniques, consider the previous theorem:

```
type T = a,b(next: T) | c;
tree x: T;
∀pos p:x.(p=a ⇒ ∃pos q:x.(p<q ∧ q=c))
```

We have already used a number of syntactic conveniences here. From the above specification, the compiler inserts the necessary coercions to reconstruct the more explicit code:

```
type T = a,b(next: T) | c;
tree x: T;
∀pos p:x,T.(read(p)=T:a ⇒
        ∃pos q:x,T.(p<q ∧ read(q)=T:c))
```

which is somewhat harder to read. In a real-life 12-page formula, more than 400 such pedantic corrections are automatically performed.

Decompilers

Any compiler writer must also consider the need for decompilers. In the case of FIDO and MONA, specifications are translated into a more primitive logic. This is fine, if we only want to decide validity. However, MONA also has the ability to generate counter-examples for invalid formulas. But a MONA counter-example will make little sense for a FIDO programmer, since it will have a completely different structure and be riddled with bit patterns. Consequently, the FIDO system provides a decompiler that lifts such counter-examples into the high-level syntax.

Another use of MONA, illustrated in the following section, is to generate specific automata. For this application, FIDO provides a different decompiler that expresses an automaton as a particular kind of attribute grammar at the level of recursive data types.

5 Examples

We now provide a few examples illustrating the benefits of the FIDO notation. We include applications that aim to synthesize automata as well as some that aim to verify properties. For each case we present a toy example in some detail and sketch a large, previously published application of a similar nature.

Synthesis

The following example considers (a fragment of) the HTML syntax. Not all syntactically correct HTML-specifications should be allowed. For example, a document should never contain an anchor within another anchor (to not confuse the reader). Such a constraint could be incorporated into the context-free syntax, but it would essentially double the number of non-terminals. However, we can easily capture HTML parse trees as values of a recursive data type. On these trees we can then express as a logical formula the restriction that we wish to impose:

```
type HTML = word |
              anchor(u: URL, a: HTML) |
              bold(b: HTML) |
              italic(i: HTML) |
              paragraph |
              rule |
              list(l: LIST);
type LIST = empty |
             entity(h: HTML, next: LIST);
type URL = url;

func Restrict(tree h: HTML): formula;
  ∀pos p: h,HTML.(p=anchor ⇒
  ¬(∃pos q: h,HTML.(p<q ∧ q=anchor)))
end;
tree H: HTML;

Restrict(H)
```

Furthermore, we can introduce any number of such restrictions in a completely modular manner. From this specification, the FIDO system can produce an attribute grammar working on parse trees, which could then easily be incorporated into an HTML development system. In this case, the attribute grammar has three attribute values, corresponding to zero, one, or too many nested anchors. Only trees synthesizing the values zero or one are accepted. The transitions, which are simply inherited from the tree automaton that MONA computes, are as follows:

HTML | word:	[] ↦ 0
HTML | anchor:	[0,0] ↦ 1
	[0,1] ↦ 2
	[0,2] ↦ 2
HTML | bold:	[0] ↦ 0
	[1] ↦ 1
	[2] ↦ 2
HTML | italic:	[0] ↦ 0
	[1] ↦ 1
	[2] ↦ 2
HTML | paragraph:	[] ↦ 0
HTML | rule:	[] ↦ 0
HTML | list:	[0] ↦ 0
	[1] ↦ 1
	[2] ↦ 2
LIST | empty:	[] ↦ 0
LIST | entity:	[0,0] ↦ 0
	[0,1] ↦ 1
	[1,0] ↦ 1

```
                [1,1] ↦ 1
                [0,2] ↦ 2
                [2,0] ↦ 2
                [1,2] ↦ 2
                [2,1] ↦ 2
                [2,2] ↦ 2
URL | url:      [] ↦ 0
```

The transition HTML | anchor: [0,0] ↦ 1 means that if the node is an anchor and each of its two subtrees synthesizes the attribute value 0, then it should synthesize the attribute value 1.

These simple ideas have been exploited in a collaboration with the Ericsson telecommunications company to formalize the constraints of design architectures [10].

Verification

Two specifications, of say distributed systems, can be compared by means of the implication or bi-implication connective. Consider a simple-minded mutual exclusion protocol for two processes with a shared memory:

```
Turn: Integer range 1..2 := 1;

task body Proc1 is
begin
  loop
    a: Non_Critical_Section_1
    b: loop exit when Turn = 1; end loop;
    c: Critical_Section_1;
    d: Turn := 2
  end loop;
end Proc1;

task body Proc2 is
begin
  loop
    a: Non_critical_Section_2;
    b: loop exit when Turn = 2; end loop;
    c: Critical_Section_2;
    d: Turn := 1;
  end loop
end Proc2
```

The FIDO specification models all valid interleaved computations and simply asks whether the safety property holds:

```
type Turn = [1..2];
type PC = a,b,c,d;
type State = PC & PC & Turn;
type Computation = State(next: Computation) | done;
string α: Computation;
func Trans(dom s,t: State): formula;
  let dom pc: PC; dom r: Turn.(
    trans(s,t)
      [a,pc?,r?] ↦ [b,pc?,r?] |
      [b,pc?,1] ↦ [c,pc?,1] |
      [b,pc?,2] ↦ [b,pc?,2] |
      [c,pc?,r?] ↦ [d,pc?,r?] |
      [d,pc?,r?] ↦ [a,pc?,2] |
      [pc?,a,r?] ↦ [pc?,b,r?] |
      [pc?,b,2] ↦ [pc?,c,2] |
      [pc?,b,1] ↦ [pc?,b,1] |
      [pc?,c,r?] ↦ [pc?,d,r?] |
      [pc?,d,r?] ↦ [pc?,a,1]
    end
  )
end;

func Valid(string x: Computation): formula;
  x=[a,a,1];
  ∀pos p: x.(
    if p.next≠done then
      let dom s,t: State.
      (p=s?; p.next=t?; Trans(s,t))
    end
  )
end;

func Mutex(string x: Computation): formula;
  ∀pos p: x.(p≠[c,c,?])
end;

Valid(α) ⇒ Mutex(α)
```

The formula **trans**(s,t) ... **end** denotes the binary relation on State domain values that hold for the pairs of values that can simultaneously match one of the listed cases.

The corresponding raw MONA formula looks like:

```
((ex1 [UNI_alpha] p: (root (p,[p]) & (all1 [UNI_alpha] q: (
(p <= q + 0) => ( ((q notin G0) & (q <= q.0 - 1)) | (((((((
q in G0) & (q notin S0)) & (q notin S1)) & (q notin S2)) &
(q notin S3)) & (q notin S4)) & (q = q.0))))))) => ( ((ex1
[UNI_x] POS26: (root (POS26,[POS26]) & ((POS26 notin G0) &
(((((POS26 notin S0) & (POS26 notin S1)) & (POS26 notin S2))
& (POS26 notin S3)) & (POS 26 notin S4))))) & (all1 [UNI_x
] POS_p: ((all1 [UNI_x] POS31: ((((POS_p in G 0) | (POS31
!= POS_p.0)) & ((POS_p notin G0) | (POS31 != POS_p))) | (PO
S31 n otin G0))) => (ex1 [UNI_x] POS41: (((POS_p notin G0)
& ((((POS_p notin G0) & (POS41 = POS_p.0)) | ((POS_p in G0)
& (POS41 = POS_p))) & (POS41 notin G0))) & (ex0 s0_pc,s1_pc
: (ex0 s0_r: ((((((((((((((POS_p in S0) <=> s0_pc) & ((
POS_p in S1) <=> s1_p c)) & (POS_p in S2)) & (POS_p in S3))
& ((POS_p in S4) <=> s0_r)) & ((((((POS41 in S0) <=> s0_pc)
& ((POS41 in S1) <=> s1_pc)) & (~(POS41 in S2))) & (~(POS
41 in S3))) & (~(POS41 in S4)))) | (((((((POS_p in S0) <=>
s0_pc) & ((POS_p in S1) <=> s1_pc)) & (~(POS_p in S2))) & (
POS_p in S3)) & ((POS_p in S4) <=> s0_r)) & ((((((POS41 in
S0) <=> s0_pc) & ((POS41 in S1) <=> s1_pc)) & (POS41 in S2)
) & (POS41 in S3)) & ((POS41 in S4) <=> s0_r)))) | (((((((P
OS_p in S0) <=> s0_pc) & ((POS_p in S1) <=> s1_pc)) & (POS_p
in S2)) & (~(POS_p in S3))) & (~(POS_p in S4))) & ((((((
POS41 in S0) <=> s0_pc) & (s1_t <=> s1_pc)) & (POS41 in S2)
) & (~(POS41 in S3))) & (~(POS41 in S4))))) | (((((((POS_p
in S0) <=> s0_pc) & (s1_s <=> s1_pc)) & (POS_p in S2)) & (
~(POS_p in S3))) & (POS_p in S4)) & ((((((POS41 in S0) <=>
s0_pc) & ((POS41 in S1) <=> s1_pc)) & (~(POS41 in S2))) & (
POS41 in S3)) & (POS41 in S4)))) | (((((((POS_p in S0) <=>
s0_pc) & ((POS_p in S1) <=> s1_pc)) & (~(POS_p in S2))) &
(~(POS_p in S3))) & ((POS_p in S4) <=> s0_r)) & ((((((POS41
in S0) <=> s0_pc) & (s1_t <=> s1_pc)) & (POS41 in S2)) &
(~(POS41 in S3))) & ((POS41 in S4) <=> s0_r)))) | ((((((POS
_p in S0) & (POS_p in S1)) & ((POS_p in S2) <=> s0_pc)) & (
(POS_p in S3) <=> s1_pc)) & ((POS_p in S4) <=> s0_r)) & (((
((~(POS41 in S0)) & (~s 1_t)) & ((POS41 in S2) <=> s0_pc))
& ((POS41 in S3) <=> s1_pc)) & (POS41 in S4)))) | ((((((~(
POS_p in S0)) & s1_s) & ((POS_p in S2) <=> s0_pc)) & ((POS_
p in S3) <=> s1_pc)) & ((POS_p in S4) <=> s0_r)) & (((((POS
41 in S0) & s 1_t) & ((POS41 in S2) <=> s0_pc)) & ((POS41 in
S3) <=> s1_pc)) & ((POS41 in S4) <=> s0_r)))) | (((((s0_s
& (~(POS_p in S1))) & ((POS_p in S2) <=> s0_pc)) & ((POS_p
in S3) <=> s1_pc)) & (POS_p in S4)) & (((((POS41 in S0) &
(~ (POS41 in S1))) & ((POS41 in S2) <=> s0_pc)) & ((POS41
in S3) <=> s1_pc)) & (POS41 in S4)))) | ((((((POS_p in S0)
& (~s1_s)) & ((POS_p in S2) <=> s0_pc)) & ((POS_p in S3)
=> s1_pc)) & (~(POS_p in S4))) & (((((~(POS41 in S0)) & (
POS41 in S1)) & ((POS41 in S2) <=> s0_pc)) & ((POS41 in S3)
=> s1_pc)) & (~(POS41 in S4))))) | ((((((~(POS_p in S0)) &
(~s1_s)) & ((POS_p in S2) <=> s0_pc)) & ((POS_p in S3) <=>
s1_pc)) & ((POS_p in S4) <=> s0_r)) & (((((POS41 in S0) &
(~(POS41 in S1))) & ((POS41 in S2) <=> s0_pc)) & ((POS41 in
S3) <=> s1_pc)) & ((POS41 in S4) <=> s0_r)))))) => (all1 [
UNI_x] POS_p: (((((POS_p in S0) | (POS_p notin S1)) | (POS_p
in S2)) | (POS_p notin S3)) | (POS_p in G0))))))))))
```

Since the simplistic mutual exclusion protocol is clearly correct, this formula is a tautology. However, if we mistakenly tried to verify that Proc2 could never enter the critical region:

```
func Mutex(string x: Computation): formula;
  ∀pos p: x.(p≠[?,c,?])
end;
```

then FIDO would generate the counterexample:

```
alpha = Computation:[a,a,1](
        Computation:[b,a,1](
        Computation:[b,b,1](
        Computation:[c,b,1](
        Computation:[d,b,1](
        Computation:[a,b,2](
        Computation:[a,c,2](
        Computation:done)))))));
```

which exactly describes such a computation.

For more realistic examples, internal events can be projected away by means of the existential quantifier. In [8], a detailed account is given of an application of the FIDO language to a verification problem posed by Broy and Lamport in 1994. The distributed systems are described in an interval logic, which is easily defined in FIDO. The evolution of a system over a finite segment of time is modeled as a recursive, linear data type with a constructor that define the current event. Thus position variables denote time instants. The thousands of events possible in the distributed systems that are compared are described by the types:

```
type Value = initVal,1;
type Loc = l0,l1;
type Ident = id0,id1;
type ValTag = MemVals,error;
type LocTag = MemLocs,error;
type TVal = Value & ValTag;
type TLoc = Loc & LocTag;
type Flag = normal,exception;
type RetFlag = BadArg,MemFailure;
type RpcFlag = RPCFailure,BadCall | RetFlag;
type Visible = observable,internal;
type ProcVal = ReadProc,WriteProc;
type ProcTag = procVal,error;
type TProc = ProcVal & ProcTag;
type NumArgs = n1,n2;
type Args = TLoc & TVal;
type Opn = rd,wrt;
type Mem = Opn & Loc & Value & Flag & Ident;
type Read = TLoc & Ident & Visible;
type Write = TLoc & TVal & Ident & Visible;
type Ret = TVal & Flag & RetFlag & Ident & Visible;
type RmtCall = TProc & NumArgs & Args & Ident;
type RpcRet = TVal & Flag & RpcFlag & Ident;
type Event = Mem | Read | Write | Ret | RmtCall |
             RpcRet | Tau;
type Comp = Event(next: Comp) | Empty;
```

The property to be verified requires 12 pages of FIDO specification which translates into an M2L formula of size 500,000 characters.

An entirely different use of FIDO allows us to verify many properties of PASCAL programs that use pointers [7]. By encoding a store as a string and using FIDO formulas to describe the effects of program statements, we can automatically verify some desirable properties. An example is the following program, which performs an in-situ reversal of a linked list with colored elements:

```
program reverse;
type Color = (red,blue);
     List = ^Item;
     Item = record
              case tag: Color of
              red,blue: (next: List)
            end;
var x,y,p: List;
```

```
begin
    while x<>nil do
    begin
       p:=x^.next;
       x^.next:=y;
       y:=x;
       x:=p
    end
end.
```

With our system, we can automatically verify that the resulting structure is still a linked list conforming to the type `List`. We can also verify that no pointer errors have occurred, such as dangling references or unclaimed memory cells. However, we cannot verify that the resulting list contains the same colors in reversed order. Still, our partial verification will clearly serve as a finely masked filter for many common programming errors.

The PASCAL tool adds another level of compilation, from (simple) PASCAL programs to FIDO specifications to M2L formulas and finally to finite-state automata accepting encodings of the initial stores that are counterexamples. The above program translates into 10 pages of FIDO specification which expands into a 60,000 character M2L formula. The resulting automaton is of course tiny since there are no counterexamples, but the largest intermediate result has 74 states and 297 BDD-nodes. A direct translation into MONA would essentially add all the complexities of the FIDO compiler to the implementation of the PASCAL tool.

6 The Implementation

We have implemented parsing, symbol analysis, and type checking in entirely standard ways. What is non-standard is that every subterm is compiled into a tree automaton through an intermediate representation as an M2L formula. Thus resource allocation becomes a question of managing bit pattern encodings of domain values, which are expressed in M2L formulas. We have strived to achieve a parsimonious strategy, since every bit squandered may potentially double the MONA execution time.

As a concrete example, consider the type:

```
type Tree = red,black(val: Enum,
                      left,right: Tree) |
            leaf;
type Enum = [1..10];
```

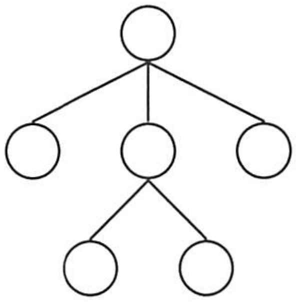
Its encoding in MONA requires seven bits in all. Two *type bits* `T0` and `T1` are used to distinguish between the types Tree and Enum and special null nodes in a tree; a single *group bit* `G0` is used to distinguish between the red-black and the leaf variants; and four *scalar bits* `S0`, `S1`, `S2`, and `S3` are used to distinguish between the values of each final domain, the largest of which is [0..10].

As an example, the formula:

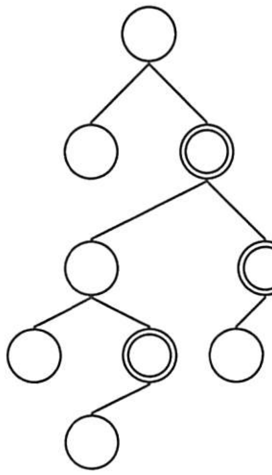
```
macro TYPE_Tree(var1 p) =
  (p in T0) & (p notin T1);
```

expresses that the type Tree is encoded by the bit pattern 10.

The null nodes are required to encode an arbitrary fan-out in a binary tree. For example, the tree:



is represented as:



where the null nodes have double lines.

A well-formed value of the type Tree is described by the MONA predicate `TREE_Tree`. It imposes the proper relationship between types and values of nodes and their descendants. A technical problem is that this predicate is most naturally described through recursion which is not available in M2L. This is solved by phrasing the requirements through a universal quantification that imposes sufficient, local well-formedness properties:

```
macro TREE_Tree(var1 p) =
  TYPE_Tree(p) &
  (all1 q: (p<=q) =>
          (NULL(q) | WF_Tree(q) | WF_Enum(q))
  );
```

The NULL and WF predicates describe the relationship between a single node and its immediate descendants:

```
macro NULL(var1 p) =
  (p notin T0) & (p notin T1) &
  (p notin G0) &
  (p notin S0) & (p notin S1) &
  (p notin S2) & (p notin S3);

macro TYPE_Enum(var1 p) =
  (p notin T0) & (p in T1);

macro GROUP_Tree_red_black(var1 p0) =
  (p notin G0);

macro GROUP_Tree_leaf(var1 p) =
  (p in G0);

macro GROUP_Tree(var1 p) =
  GROUP_Tree_red_black(p) | GROUP_Tree_leaf(p);

macro SCALAR_Enum(var1 p) =
  (p notin S3) |
  ((p notin S2) & ((p notin S1) | (p notin S0)));

macro SCALAR_Tree_red_black(var1 p) =
  true;

macro SCALAR_Tree(var1 p) =
  SCALAR_Tree_red_black(p);

macro SUCC_Enum(var1 p) =
  (p=p.0) & (p=p.1);

macro SUCC_Tree_red_black(var1 p) =
  (p<p.0) & (p<p.1) & (p.1<p.11) & (p.11=p.111) &
  NULL(p.1) & NULL(p.11) &
  TYPE_Tree(p.0) & TYPE_Tree(p.10) &
  TYPE_Enum(p.110);

macro SUCC_Tree_leaf(var1 p) =
  (p=p.0) & (p=p.1);

macro WF_Enum(var1 p) =
  TYPE_Enum(p) & SCALAR_Enum(p) & SUCC_Enum(p);
```

```
macro WF_Tree(var1 p) =
  TYPE_Tree(p) &
  ((GROUP_Tree_red_black(p) &
    SCALAR_Tree_red_black(p) &
    SUCC_Tree_red_black(p)
   ) |
   (GROUP_Tree_leaf(p) &
    (p notin S0) & SUCC_Tree_leaf(p)
   )
  );
```

Formulas are encoded in a simple inductive manner. For illustration, consider the tiny formula p∈s, where the arguments are general terms. The term p of kind **pos** generates a tuple $< p, \phi >$ where p is a first-order variable constrained by the formula ϕ. Similarly, the term s of kind **set** generates a tuple $< s, \psi >$, where s is now a second-order variable. The term p∈s then generates the formula $\exists p : \exists s : \phi \wedge \psi \wedge p \in s$. Note how existential quantification corresponds to discharging of registers. It is a fairly straightforward task to provide similar templates for all the FIDO constructs, thereby providing a compositional semantics and a recipe for a systematic translation.

As a concrete example, consider the formula:

tree x: Tree;
x.left.right.left=red

which describes the regular set of trees in which a specific node exists and is colored red. It is encoded as the following MONA formula:

```
macro DOT_right(var1 p,var1 q) =
  (TYPE_Tree(p) &
   GROUP_Tree_red_black(p) & (q=p.0)
  ) |
  (TYPE_Tree(p) &
   GROUP_Tree_leaf(p) & (q=p)
  );

macro DOT_left(var1 p,var1 q) =
  (TYPE_Tree(p) &
   GROUP_Tree_red_black(p) & (q=p.10)
  ) |
  (TYPE_Tree(p) &
   GROUP_Tree_leaf(p) & (q=p)
  );

assume ex1 p: root(p) & TREE_Tree(p);

ex0 t0_1,t1_1,g0_1,s0_1:
ex0 t0_2,t1_2,g0_2,s0_2:
```

```
(ex1 POS6:
  (ex1 POS5:
    (ex1 POS4:
      (ex1 POS3:
        root(POS3) & DOT_left(POS3,POS4)
      ) &
      DOT_right(POS4,POS5)
    ) &
    DOT_left(POS5,POS6)
  ) &
  (t1_1<=>(POS6 in T1)) &
  (t0_1<=>(POS6 in T0)) &
  (g0_1<=>(POS6 in G0)) &
  (s0_1<=>(POS6 in S0)) &
  (t0_2 & ~t1_2 & ~g0_2 & ~s0_2) &
  (g0_1 <=> g0_2) & (s0_1 <=> s0_2)
);
```

The analogy to run-time is the computation by MONA of a finite-state automaton from the generated formula. This is always guaranteed to terminate, but may be prohibitively expensive. Thus, the FIDO compiler does extensive optimizations at many levels, in most cases relying heavily on the type structure. FIDO formulas are symbolically reduced to detect simple tautologies and to eliminate unnecessary variables and quantifiers. A careful strategy is employed to allocate short bit patterns for finite domains, which includes a global analysis of concrete uses.

We have also discovered that the FIDO type structure contains a wealth of information that is not currently being exploited by the MONA implementation. An ongoing development effort will enrich the notion of tree automata to accommodate positional information that can be derived from FIDO specifications. This may in some case yield an exponential speed-up at the MONA level.

7 FIDO as a DSL

In our opinion, FIDO is a compelling example of a domain-specific language. It is focused on a clearly defined and narrow domain: *formulas in monadic second-order logic* or, equivalently, *automata on large alphabets.* It offers solutions to a classical software problem: *drowning in a swamp of low-level encodings.* It advocates a simple design principle: *go by analogy to standard programming language concepts.* It uses a well-known and trusted technology: *all the phases of a standard compiler, including optimizations at all levels.* It provides unique benefits that cannot be matched by a library in a standard programming language: *notational conveniences, type checking, and global optimizations.* And during its development, we discovered new insights about the domain: *new notions of tree automata and algorithms.*

References

[1] A. Ayari, D. Basin, and A. Podelski. LISA: A specification language based on WS2S. In *Proceedings of CSL'97.* BRICS, 1997.

[2] D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Computer aided verification : 7th International Conference, CAV '95, LNCS 939*, 1995.

[3] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *Proceedings of WIA'96.* Springer Verlag, 1996.

[4] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, August 1986.

[5] Jesper Gulmann Henriksen, Michael Jrgensen, Jakob Jensen, Nils Klarlund, Bob Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of TACAS'95, LNCS 1019*, May 1995.

[6] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, May 1997. Special issue on Formal Methods in Software Practice.

[7] J.L. Jensen, M.E. Jrgensen, N. Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of PLDI'97*, 1997.

[8] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in automated verification based on trace abstractions. Technical Report RS-95-54, BRICS, Aarhus University, 1995.

[9] N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstraction. In *Proceedings of PODC'96*, 1996.

[10] Nils Klarlund, Jari Koistinen, and Michael I. Schwartzbach. Formal design constraints. In *Proceedings of OOPSLA'96*, October 1996.

[11] Nils Klarlund and Theis Rauhe. BDD algorithms and cache misses. Technical Report RS-96-05, BRICS, 1996. Submitted.

[12] J.C. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development, LNCS 185*, 1985.

A Modular Monadic Action Semantics*

Keith Wansbrough *John Hamer*

Department of Computer Science
University of Auckland
Auckland, New Zealand
{*keith-w, j-hamer*}@*cs.auckland.ac.nz*

Abstract

A domain-specific language (DSL) is a framework which is designed to precisely meet the needs of a particular application. Domain-specific languages exist for a variety of reasons. As productivity tools, they are used to make application prototyping and development faster and more robust in the presence of evolving requirements. Furthermore, by bridging the "semantic gap" between an application domain and program code, DSLs increase the opportunity to apply formal methods in proving properties of an application.

In this paper, we contribute a synthesis of two existing systems that address the problem of providing *sound* semantic descriptions of *realistic* programming languages: action semantics and modular monadic semantics. The resulting synthesis, modular monadic action semantics, is compatible with action semantics yet adds true modularity and allows domain specific specifications to be made at a variety of levels.

1 Introduction

"I'd rather write programs to write programs than write programs." (Programming proverb).

A domain-specific language (DSL) is a framework which is designed to precisely meet the needs of a particular application. Domain-specific languages exist for a variety of reasons. As productivity tools, they are used to make application prototyping and development faster and more robust in the presence of evolving requirements. Furthermore, by bridging the "semantic gap" between an application domain and program code, DSLs increase the opportunity to apply formal methods in proving properties of an application.

Designing and implementing a domain specific language is, however, problematic. Putting aside the requirement for an efficient implementation, the language needs to be: clearly and precisely defined; modular enough to change incrementally; and amenable to formal reasoning. Different tools exist that address these varied requirements, but few make any attempt to address them all. Compiler generators provide efficient implementations, but they typically have weak formal properties. We do not consider such systems further in this paper. Semantic formalisms, such as denotational semantics and structural operational semantics, have richly developed theories but are difficult to use and lack modularity.

The conceptual distance between the high-level language and established semantic formalisms is huge. Writing such specifications is a tedious and difficult task. Natural concepts in the high-level language must be accurately simulated by constructs built from the small range of primitives in the semantic formalism. The probability of introducing errors is large, and the maintainability of the specification is poor.

Three attempts at solving the problem of providing *sound* semantic descriptions of *realistic* programming languages are presented here. The first is *action semantics* [Mos92], a highly readable notation with formal foundations in Peter Mosses's unified algebra and Plotkin's structural operational semantics [Plo83]. This system has proven quite popular, and has been used to specify a number of existing and evolving languages.

The second is Hudak, Liang and Jones' *modular monadic semantics* [LH96]. Modular monadic semantics is a structured form of denotational semantics, embedded in the Haskell language. As a relatively new system it is yet to gain widespread use, but it has a number of highly attractive features, foremost its excellent modularity properties.

These two systems can be seen as competing for the same market (Figure 1(c) and (d)): both are attempts to reduce the conceptual distance that must be bridged when formally specifying a high-level language (compare with Figure 1(a) and (b)). Both appear to do so successfully; they are, however, different DSLs and are based in different semantic formalisms.

*This work was supported in part by a scholarship from the University of Auckland.

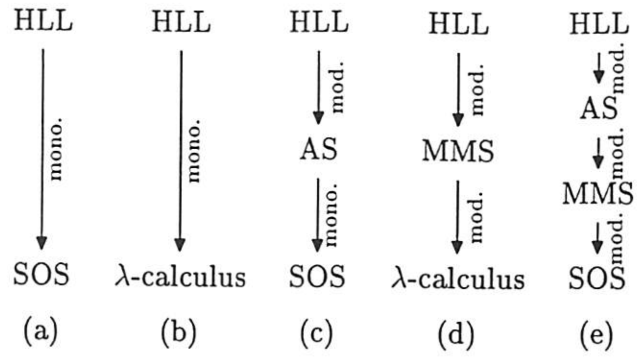


Figure 1: Formal specification techniques.

The third approach arises our observation that action semantics and modular monadic semantics have much to offer each other. Action semantics provides a friendly syntax to the language specifier, and encapsulates a substantial but fixed range of language concepts. Modular monadic semantics, on the other hand, provides a rather less friendly syntax, but with complete flexibility and extensibility afforded by its internal modularity. By combining these two systems, we obtain a tiered system containing *two* domain-specific semantic notations, featuring all the key features of action semantics along with the underlying flexibility and extensibility of modular monadic semantics (Figure 1(e)). This system we call *modular monadic action semantics* (MMAS).

The user of modular monadic action semantics specifies the high-level language using the rich set of primitives provided by action semantics. If the language requires a concept that is not present in action semantics, the specifier simply drops down one level, and uses the underlying modular monadic semantics to specify the new feature. If this is not sufficient, the modular monadic layer itself may be modified or extended as required. All this takes place in a highly modular fashion, with little or no change to existing parts of the specification. Furthermore, *theories* developed at each layer can also be developed in a modular fashion. New features can be introduced without undermining the validity of existing proofs.

In the remainder of this paper we describe action semantics, modular monadic semantics, and modular monadic action semantics in more detail, and we investigate some of the applications of MMAS. We conclude with a look at some related work, some concluding comments, and directions for future work. Full details on modular monadic action semantics can be found in Wansbrough's thesis [Wan97].

2 Action semantics

Action semantics [Mos92, Mos96a] is a notation developed over some years by Mosses at Aarhus University. As Mosses puts it,

> The primary aim of action semantics is to allow *useful* semantic descriptions of *realistic* programming languages. [Mos92, p. xv]

It aims at being simultaneously formal and readable, and achieves this goal admirably. An action semantic description is entirely formal, yet it can be intuitively understood to a surprising degree by someone with no prior knowledge of the system.

However, action semantics has two significant limitations. Firstly, it is incomplete. Not all programming language concepts can be represented directly within action semantics—only those which Mosses has chosen to build into the system. There is no provision for the extension of action semantics. Secondly, action semantics theory has not progressed very far: it is difficult to use an action semantics of a language to prove properties of that language or of programs written within it.

Despite these disadvantages, action semantics has proven quite popular and rather successful. A number of compiler generators based on action semantics exist [Ørb94, BMW92, Pal92], along with at least one tool [vDM96] for assistance in developing new action semantic descriptions. Action semantic descriptions exist for a number of real languages [Mos96a, §7.1], and others are being developed. An action semantics for an intermediate language for compilers, ANDF-FS [NT94, HT94], is currently in use in the 'real world', by the Open Software Foundation. Work is progressing in a number of directions.

In the following sections the essential features of action semantics are briefly summarised. More detail can be found in Mosses' book [Mos92], or his tutorial [Mos96b].

2.1 Structure of action semantics

An action semantic description (ASD) of a language specifies the semantics of the language by means of *actions*, representing computations. The action corresponding to a given program phrase is built up from primitive actions and action *combinators*. Data referred to by the actions may be accessed by means of *yielders*, and new types of data may be readily defined in an algebraic fashion.

Action semantics divides program behaviour into a number of *facets*, corresponding to the types of information dealt with. The *basic* facet refers to control flow. The *functional* facet refers to transient information, i.e., data passed between successive actions. The *declarative* facet refers to scoped information, i.e., bindings of tokens to data. The *imperative* facet refers to stable information, i.e., the storage of data in cells. Finally, the *communicative* facet refers to permanent information, i.e., the communication of data between distributed actions. Two further facets, the *reflective* and *directive* facets, refer to reflection and indirection. Action semantics has these seven computational concepts built in, and they can be used directly by specifications.

The primitive actions, yielders and data of action semantics are separated into these categories, and are intended to act on only one facet at a time. This automatically lends a degree of modularity to an action semantic description: actions involving only, say, basic and functional behaviour (such as expression evaluation) need not concern themselves with behaviour in other facets (such as the declarative or imperative facets). The result is a readily maintainable description: alterations to behaviour relating to one facet do not affect unrelated code (as they inevitably do when a monolithic approach is used, such as the conventional structural operational semantic or lambda-calculus approaches).

A key advantage of action semantics is its very readable notation. Action semantics uses words rather than symbols, and these are chosen in a way that allows even a reader unfamiliar with action semantics to obtain a broad impression of the intended meaning of the description. This is partially enabled by its flexible (albeit idiosyncratic) type system, which permits definition of new types by extending or specialising existing ones and definition of abbreviations for commonly-occurring patterns of notation, and behaves much more like conventional set theory than do traditional domains.

As a simple example of the use of action semantics, consider the following semantics for the 'if' statement of a conventional imperative language. In this example, execute is a semantic function which gives the semantics of statements. The brackets $[\![\cdot]\!]$ enclose the abstract syntax for the statement, and the term to the right of the equals sign is the action representing the semantics of the statement.

```
execute [[ "if" $E$ "then" $S_1$ "else" $S_2$ ]] =
    evaluate $E$ then
    | | check it and then execute $S_1$
    | or
    | | check not it and then execute $S_2$ .
```

The action corresponding to the phrase "if" E "then" S_1 "else" S_2 begins by calling evaluate E, defined elsewhere. The result of the expression is threaded by the then combinator to the next action. This is an or, which nondeterministically executes one branch and backtracks on failure. The action check, applied to the yielder it, fails if the yielded value is false and succeeds if it is true; the other branch has the complementary test. After the test, execution passes (via and then) to an execute of the appropriate branch.

This example shows the way in which action semantics builds up actions from primitive actions, combinators, and yielders. Notice how easy the specification is to read, even without any familiarity with the notation. Also note that there is no need to thread the store through this equation, even though the evaluation of E may access or modify it. The modularity of action semantics means we need not consider facets not directly referenced.

2.2 Semantics

Action semantics is intended to be a formal notation, and in order to achieve this its own semantics must be precisely and formally defined. Mosses chose to do this by providing a low-level definition of action semantics as a *structural operational semantics* (SOS) [Plo81, Plo83], using a slight variation on Plotkin's original approach.

As an example, here are some of the equations describing the behaviour of the or combinator, used above to define the semantics of the 'if' statement. The equations are taken from [Mos92, §C.3.3.2.1].

(7) stepped $(A_1, l) \geq (A'_1$:Acting, l':local-info, uncommitted) ;
$[\![A_1\ O\ A_2]\!]$: $[\![$ Intermediate "or" Intermediate $]\!] \Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info)
$\geq$ (simplified $[\![A'_1\ O\ A_2]\!]$, l', uncommitted) .

(8) stepped $(A_2, l) \geq (A'_2$:Acting, l':local-info, uncommitted) ;
$[\![A_1\ O\ A_2]\!]$: $[\![$ Intermediate "or" Intermediate $]\!] \Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info)
$\geq$ (simplified $[\![A_1\ O\ A'_2]\!]$, l', uncommitted) .

(9) stepped $(A_1, l) \geq (A'_1$:Acting, l':local-info, c':committing) ;
$[\![A_1\ O\ A_2]\!]$: $[\![$ Intermediate "or" Intermediate $]\!] \Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info) $\geq (A'_1, l', c')$.

(10) stepped $(A_2, l) \geq (A'_2$:Acting, l':local-info, c':committing) ;
$[\![A_1\ O\ A_2]\!]$: $[\![$ Intermediate "or" Intermediate $]\!] \Rightarrow$
stepped ($[\![A_1\ O\ A_2]\!]$, l:local-info) $\geq (A'_2, l', c')$.

Equations (7) and (8) handle uncommitted cases, and equations (9) and (10) committed cases (commitment is analagous to the action of Prolog's 'cut' operator (!): it commits to the chosen branch, prohibiting backtracking). In both cases, the equations specify nondeterministic choice of one action to be stepped. Once this is done, if the state remains uncommitted then equations (7) and (8) indicate a simplification is to be performed over the whole phrase; if the state becomes committed then equations (9) and (10) specify that the alternate branch is discarded. The result (in combination with the omitted definition of the simplify function) is a nondeterministic backtracking choice with commitment.

2.3 Problems

Action semantics provides an usable foundation on which to build specifications of programming languages. Referring back to Figure 1, it bridges the gap between high-level language and low-level formalism very effectively, providing a language that already contains the features required by the high-level language. In addition, it is highly readable and ensures a degree of modularity in specifications.

However, the features supported by action semantics are fixed: the facets of action semantics are those defined by Mosses, and no mechanism is provided to modify or extend them. The facets provided are sufficient to describe many common imperative languages; but there are language features (notably continuations, as popularised by Scheme for example) that are *not* present in action semantics. The only way to specify languages containing such features is by low-level simulation; resorting to this instantly loses all the advantages of action semantics.

One of the reasons for the fixed nature of the facets in action semantics is indicated in the diagram in Figure 1(c). While the specification of the high-level language in action semantics is done in a modular manner, Mosses' definition of action semantics in terms of structural operational semantics is done monolithically. This means that there is no real separation between facets at this level, and the defining equations for each component must correctly handle not only their own information but also the information from other, unrelated components. Without modularity we are back to the situation of Figure 1(a), where specification is tedious and error-prone, and maintainability is poor.

Another problem with action semantics lies in its theory. One of the goals of a formal specification of a language, as articulated by Mosses [Mos92, p. 4], is to use it

> ... as a basis for *reasoning* about the correctness of particular programs in relation to their specifications, and for justifying program transformations.

An action semantic description of a language certainly provides a formal specification, but is it useful? Mosses himself notes that "a decent theory for action semantics has been slow to emerge" [Mos96a, §6]. Compilers incorporating provably valid action transformations exist [Mos96a, §6.2–3]; but the closest approach to a general and tractable theory for action equivalence still covers only the basic, functional and declarative facets of action semantics—omitting the imperative and communicative facets essential to most real programming languages [Mos96a, §6.4].

3 Modular monadic semantics

Modular monadic semantics (MMS) [LHJ95, LH96] is a structured form of denotational semantics developed recently by Liang, Hudak, and Jones of Yale University based on the work of Moggi [Mog89a, Mog91a] and Espinosa [Esp93, Esp94]. It provides a very usable approach to denotational semantics with excellent flexibility and modularity properties. MMS is presented in the syntax of an existing functional language, thus permitting specifications to be directly executed.

In the following sections we very briefly note the essential features of modular monadic sematics; more details may be found in [LH96] and [LHJ95].

3.1 Structure of MMS

Modular monadic semantics specifies the semantics of a language by a mapping from terms to *computations* performed within a *monad*. The monad hides details of semantic features such as environments and stores that are used by the computation, and exposes operators allowing access to these features.

All monads have the two primitive operators $return$ and $>\!\!>=$ (pronounced "bind"). The expression $return\ x$ represents the trivial computation with result x; the expression $c_1 >\!\!>= \lambda v \rightarrow c_2$ represents the computation that computes c_1, binds the result to v, then computes c_2.

In addition to these essential operators, monads encapsulating semantic features provide operators to access them. For example, an environment monad might provide $rdEnv$ and $inEnv$ operators; a continuation monad would provide a *callcc* operator.

This abstraction allows the underlying monad (representing the required semantic features) to be modified without altering the specification that uses it. Even if new features are added to the monad, the existing interface remains unchanged. Equations from one specification can be used within another, as long as the features used in the one are all found in the other.

The abstraction also allows us to ignore irrelevant details. As we manipulate the store, for example, we need not concern ourselves with preserving the environment—this is done transparently by the $return$ and $>\!\!>=$ operators.

Alone, this is not sufficient. Even though the specification using the monad need not change as the feature set represented changes, it is clear that the monad itself must change: it must incorporate new operators, and the behaviour of existing operators (especially $return$ and $>\!\!>=$) must change appropriately. For this reason, we introduce *monad transformers*.

A *monad transformer* is an object that transforms a monad, modifying the behaviour of its existing operators and adding new ones. In modular monadic semantics, we represent each desired semantic feature by a monad transformer, and then apply them all to a trivial monad. The resulting monad incorporates all of the features of the component monad transformers, and is used for the semantic specification of the high-level language.

The great advantage of this system its flexibility. Depending on how much support is needed, any set of monad transformers representing any set of semantic features may be combined in a modular fashion to provide that support. Monad transformers provide the power needed to encapsulate high-level semantic features, but still allow access to the low-level semantic detail.

The system is highly modular: the definition of each monad transformer is independent of the others; one need not concern oneself with handling the details of unrelated semantic features. As well as simplifying the writing of monad transformers, this also means that one may construct modular *proofs* within the system: as modular monadic semantics is simply a structured form of denotational semantics our normal proof methods still apply, but now we may deal with each semantic feature separately and rely on the system to cleanly and safely combine them.

3.2 Example

As a simple example of the use of modular monadic semantics, consider a specification for a small expression language. The language is to support nested environments and exceptions. These are common language features, and from a standard library of monad transformers we select *ErrorT* for supporting errors and *EnvT* for environments. A monad for our expression language can be formed by composing these monad transformers as follows[1]:

$$\mathbf{type}\, M \;=\; ErrorT\;(EnvT\;Id)$$

Next, consider the equation for the addition operation:

$$\begin{aligned} &evaluate\;(Add\;e_1\;e_2) = \\ &\quad evaluate\;e_1 \gg\!= \lambda v_1 \to \\ &\quad evaluate\;e_2 \gg\!= \lambda v_2 \to \\ &\quad return\;(v_1 + v_2) \end{aligned}$$

Here we simply evaluate the two subexpressions and return the sum of the results. Errors and environments are transparently passed around by the >>= operator, and hence need not be considered at all by this equation.

The equation for a variable reference involves both environments and errors, and demonstrates the use of the operators provided by those monad transformers:

$$\begin{aligned} &evaluate\;(Var\;x) = \\ &\quad rdEnv \gg\!= \lambda\rho \to \\ &\quad \mathbf{case}\; lookup\; x\;\rho\;\mathbf{of} \\ &\qquad Just\;v \to return\;v \\ &\qquad Nothing \to raise\;\text{"unbound identifier"} \end{aligned}$$

Here *rdEnv* and *raise* are operators provided by the environment and error monad transformers, respectively. The function *lookup* is a helper function defined elsewhere. The environment is accessed by means of the operator *rdEnv*, "read the value of the environment", which uses the environment that is passed implicitly within the monad; it does not need to appear explicitly as a parameter to *evaluate*.

Underneath this example, of course, are the actual monad transformers involved. As mentioned before, a monad transformer adds new operators to the monad, modifies the existing ones, and alters the definitions of *return* and >>=. The definition of *EnvT*, the environment monad transformer, is as follows:

$$\begin{aligned} \mathbf{type}\; EnvT\;m\;a &= e \to m\;a \\ return_{(EnvT\;m)}\;v &= \lambda\rho \to return_m\;v \\ c \gg\!=_{(EnvT\;m)} f &= \lambda\rho \to c\rho \gg\!=_m \lambda v \to f v \rho \\ rdEnv_{(EnvT\;m)} &= \lambda\rho \to return_m\;\rho \\ inEnv_{(EnvT\;m)}\;c &= \lambda\rho' \to c\rho \\ lift_{(EnvT\;m)}\;c &= \lambda\rho \to c \end{aligned}$$

[1] *Id* is the trivial monad

The first equation defines the new monad transformer *EnvT* as taking a monad m defined over a result type a and turning it into a new monad, *EnvT* m, over a. An encoding in the lambda calculus is given for environments.

Next we define the operators $return_{(EnvT\;m)}$ and $\gg\!=_{(EnvT\;m)}$, in terms of the operators of the lower monad m.

We define the operators *rdEnv* and *inEnv* next, thus permitting access to the environment now being passed around within the monad.

Finally, we define an operator $lift_{(EnvT\;m)}$ which is used to transform computations in the lower monad m to computations in the upper monad *EnvT* m. This lifting operator is used by the MMS system to lift the operators of m to the upper monad level, so they may still be used.

In reality, the situation is a little more complex than this. For details, consult Liang, Hudak, and Jones' paper [LHJ95].

4 Modular monadic action semantics

We have seen that action semantics is an excellent notation for describing the semantics of real programming languages. However, we have also seen that certain constructs pose grave difficulties when the language specifier attempts to encode them in action semantics. These difficulties are due to the fixed nature of action semantics—certain notions of computation are built into action semantics, and any that are not must be tediously simulated. This situation is inadequate.

As we noted in Section 2.3, the fixed nature of action semantics is in large part due to the monolithic nature of its underlying semantic definition, written in a variant of Plotkin's structural operational semantics.

The recent work on modular monadic semantics, described in Section 3, suggested to us a solution to the problem. Modular monadic semantics provides a mode of semantic definition that is truly modular. Yet it is also sufficiently low-level to be used for the specification of action notation. A tiered system with action notation specified by a modular monadic semantics would preserve the user-friendliness of action notation, but permit the notation to be modified relatively easily to incorporate even quite major modifications or additions to the notions of computation represented.

As an added bonus, modular monadic semantics descends ultimately from denotational semantics, and so inherits its rich theory—yet without the customary tangle of unmaintainable equations. Hence, while providing a clear operational interpretation of action notation, a modular monadic semantics for action notation would also provide a sound basis for the development of theories of action semantics.

The result of these observations is MMAS—a modular monadic action semantics. MMAS appears identical to Mosses' action semantics, but internally its semantics is specified by means of a modular monadic semantics in the style of Liang, Hudak, and Jones, rather than Mosses' structural operational semantics. As a consequence, MMAS is modular and extensible, and dialects of MMAS can be created that incorporate new or modified notions of computation.

4.1 Structure of MMAS

In Figure 1 we have considered the structure of action semantics as a formal specification technique, along with other semantic formalisms. We now consider the structure of systems that *implement* action semantics, in order to interpret or compile a language defined in it.

In Figure 2(a) we have Mosses' definitive formal model, as described in [Mos92]. The action-semantic interface of the system is described by a kernel action notation (here denoted 'KAN') and a layer of "sugar" reducing full action notation to this kernel action notation. The kernel is described in terms of a structural operational semantics (see Section 2.2), and this structural operational semantics is executed by an abstract machine.

It is not practical to implement Mosses' formal model directly. Instead such a system is simulated by some other technique, and this shown formally or informally to be equivalent to Mosses' scheme. The architecture of an action semantics interpreter of this nature is depicted in Figure 2(b). Notice that the interpreter in this system is generally more or less monolithic, and implemented in some low-level implementation language such as C.

An action compiler (or action semantics-based compiler generator) is similar, except that actual code generation is usually deferred to another compiler and hence the compiler is really a translator between action notation and the low-level implementation language. Figure 2(c) depicts this scenario. Again, the system is essentially monolithic: the program implements the semantics of action notation, but in an opaque manner that is not at all easy to modify.

In contrast to these techniques, consider the MMAS approach, as shown in Figure 2(d). Here the external action semantic interface of the system is provided by the *action notation layer*, which is coded in modular monadic semantics. The action notation layer depends on a *monad transformer layer*, which defines and combines a series of monad tranformers representing features required by the action notation layer. As there is some common code in the monad transformer layer, this is abstracted out into the *general monad transformer layer*; both these latter two layers

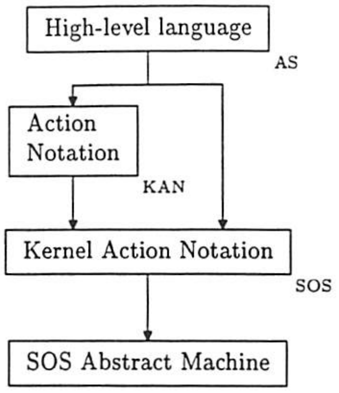


Figure 2(a): Mosses' model action semantics system

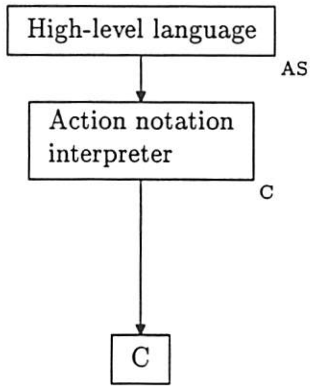


Figure 2(b): An action semantics interpreter

are coded in Haskell and the system depends on an underlying Haskell interpreter or compiler.

The larger number of distinct layers in this scheme ensures that flexibility and modularity is possible at each level. The modular structure of each layer is depicted in Figure 3. Features in the high-level language are based upon notions of computation provided by modules in the action notation layer, which are based in turn on those provided by the monad transformers of the monad transformer layer, and so on. As features change in the high-level language, the architecture allows existing modules to be removed or modified and additional modules to be added, with a high-degree of independence from other modules in the system.

4.2 The base system

Like Mosses' action semantics, modular monadic action semantics can be divided into facets. In general, each facet consists of an action notation module and a supporting monad transformer providing the notions of computation on which it relies. However, this need not be true of all facets: action notation modules and monad transformers

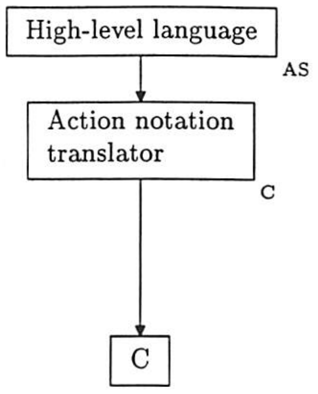


Figure 2(c): An action semantics compiler

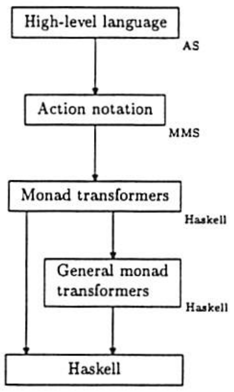


Figure 2(d): The MMAS action semantics system

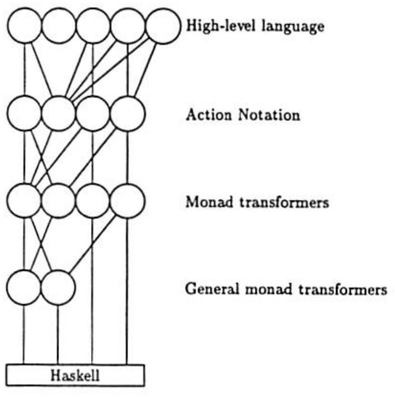


Figure 3: Detail of the MMAS action semantics system

are independent. Both an action notation module relying only upon existing monad transformers and a monad transformer with no related action notation module are possible, and indeed are present in base MMAS.

Base MMAS, the unmodified form of the MMAS system, implements almost all of Mosses' action semantics. It contains six action notation modules: basic and functional, declarative, imperative, reflective, directive, and communicative. These correspond directly to the seven facets of Mosses' action notation, except that the basic and functional facets have been merged into one.

To support this action notation layer, base MMAS contains eight monad transformers: one each for the basic and functional, declarative, imperative, directive, and communicative facets (there is none for the reflective facet), along with two others used to implement parallelism and nondeterminism (built into Mosses' structural operational semantics but not present in the lambda-calculus basis of MMAS).

Code common to a number of these monad transformers suggested the extraction of two general monad transformers, one for the representation of environments and the other for the representation of state.

This base system implements almost all of Mosses' action semantics: in most cases existing action semantic descriptions can be used with MMAS with very little modification. Such ASDs can be interpreted in an MMAS system based over a Haskell interpreter, or made into compilers using an optimising Haskell compiler. Proofs of their properties may take advantage of the true modularity provided by the modular monadic definition of action semantics (see [LH96] for a discussion of proofs in a modular monadic context).

4.3 Branching out

The key feature of modular monadic action semantics, absent in other implementations of action semantics, is *extensibility*. Modules can be modified, removed from or added to the MMAS system.

Modification Any module of the MMAS system can be modified internally without affecting any other portion of the system, as long as its external interface is not changed. In fact, new functionality may be *added*, as long as existing functionality is unaffected.

At the level of the action notation layer, new actions, combinators, yielders or sorts may be added or the precise behaviour of existing ones altered, simply by editing the modular monadic semantics code that specifies them for the facet concerned.

At the level of the monad transformer, new features may be added or the implementation of existing ones may be altered. New or altered features can then be used by the action notation module or modules above it.

As long as modifications preserve or naturally extend existing functionality, changes are completely modular. If this is not the case, changes are required only to the modules that depend on the module modified: in the case of changes to an action notation module, none (other than existing ASDs); in the case of changes to a monad transformer, only the action notation module(s) on which it depends.

Removal Naturally, modules which are not required may be removed from the MMAS system entirely. This is of course not strictly necessary—optimisations will remove reference to the unused module anyway, and proofs are modular and will not be affected by an unreferenced module—but may be desired for neatness or security.

Addition Of greater significance is the addition of new modules. It is clear that modifying existing facets or modules is not always sufficient; in certain cases one wishes to add an entirely new feature or notion of computation to the system. The procedure for doing so in MMAS (in contrast to that for existing action semantics systems) is straightforward.

First, a new monad transformer must be defined. This transformer is written in Haskell, possibly with the aid of the existing general monad transformers and support code, and encapsulates the behaviour of the new feature. A typical monad transformer can be defined in around 40 lines of code.

Next, a new module in the action notation layer is defined. This module is written using modular monadic semantics, referring to the new monad transformer and possibly others. It defines the new action notation to be used to access the newly-defined feature. Depending on how many actions, combinators and yielders are to be defined the code required varies, but a typical size would be around 30 lines of code.

Together, this module and its supporting monad transformer add the desired new feature to the MMAS system. The system can now be used in exactly the same manner as before—existing ASDs will continue to work identically, unaware of the new feature—but new action semantics descriptions may use the feature as desired.

4.4 Reasoning in MMAS

Consider again Figure 1, and the problem of reasoning about the correctness of a program or a transformation in the high-level language by means of the formal semantics.

In fig. 1 (a), a proof of a property of the high-level language is essentially a proof about states of the abstract machine upon which SOS is based. Similarly, in fig. 1 (b) a proof concerns elements of the various domains over which the relevant semantic equations are defined. In both cases, the representations we must deal with are very low-level, and *monolithic*: i.e., all information relating to the program is contained in a single object, and all of it can potentially affect the validity of the proof and must be taken into account by it. In both SOS and λ-calculus we may take steps to alleviate this to some extent, but it remains a fundamental problem.

In fig. 1 (c), Mosses' action semantics, *modular* proofs are possible at the action semantic level. This means that we may prove a property about, say, the functional behaviour of an expression without concerning ourselves with interactions from communication or from the store. Given an action theory describing the semantics of actions at the action semantics level, facet by facet, we can construct our proofs facet by facet also.

However, the assumption here is that such an action theory exists. Certainly this *should* be the case, but in general it is not. Certain properties about actions are known, and are listed in [Mos92] and elsewhere; but no general action theory yet exists (see Section 2.3). Because of this lack, proofs about high-level language properties must usually begin by constructing the action theory they need: and this construction of action theory must be performed at the SOS level, which as we saw above is monolithic rather than modular, and hence difficult. In practice, proofs about high-level languages using action semantics are almost as hard as those using a structural operational semantics directly.

The scenario in fig. 1 (d), however, is different. As in fig. 1 (c), we may construct modular proofs of high-level language properties based on the modularity of modular monadic semantics. But with modular monadic semantics, the theory we construct to support this may also be constructed modularly: our interaction with the lambda-calculus is structured in such a way that proofs involving one monad transformer cannot be affected by properties of other unrelated monad transformers. Hence we are able to realise the promise of relatively easy modular proofs of high-level language properties, even in practice when this requires the construction of new modular monadic semantic theory, since this construction is also modular.

As has already been explained, the intention of modular monadic action semantics (fig. 1(c)) is to provide action semantics with the true modularity of modular monadic semantics. To prove high-level language properties in this system, we use action theory. But, instead of constructing this theory directly and monolithically from the SOS, we construct it in terms of the relatively high-level modular theory provided by modular monadic semantics. *This* theory, in turn, is constructed, again modularly, from the lamba-calculus properties involved in each monad transformer. The result is a fully-modular, tiered system, in which one need only consider the features that are directly relevant to the property one wishes to prove—unrelated features may safely be ignored.

Work so far on modular monadic action semantics has concentrated rather on the pragmatics of the system than on the theory, but the theoretical basis is clear. Liang and Hudak [LH96] give a discussion of proofs in modular monadic semantics corresponding to the two arrows in that we used in fig. 1 (c) of Figure 1; these are essentially the lower two of the three arrows in fig. 1 (e), modular monadic action semantics.

4.5 Limitations

Modular monadic action semantics is not completely identical to Mosses' action semantics. MMAS implements only a small part of Mosses' communicative facet. Mosses' version of this facet supports general message-passing communication between multiple parallel agents. A system of this nature is of necessity quite complex, involving not merely communication but also the creation and management of the parallel processes themselves. As the whole area of parallelism in denotational semantics is currently rather turbulent [Abr96], it was felt safest to take a conservative approach and implement only a restricted form of process–user interaction, for a single process only. Parallelism within a process is supported.

The type system used by modular monadic action semantics is essentially that of the underlying Haskell system, overlaid with Liang, Hudak, and Jones' extensible union types [LHJ95]. Compared with the unified algebras used by Mosses, this system is quite restricted: types (*sorts* in Mosses' terminology) are not first-class, and the only type operations permitted are injection, projection and disjoint union. This means that those operators in action semantics dependent upon first-class sorts (such as the nondeterministic choice operator choose) have of necessity had their semantics altered. Luckily there are few of these: Mosses writes [Mos92, p. 36] "action notation does not depend much on the unorthodox features of our algebraic specification framework."

As well as being impacted by the lack of first-class sorts (and hence unbounded nondeterminism), the nondeterminism of modular monadic action semantics is restricted by its nature as an executable system. It is constrained to 'give an answer', and hence in the end just *one* of the many possible (nondeterministic) behaviours must be exhibited, unlike action semantics which merely returns the sort of all possible behaviours.

5 An example

The notion of computation that is most obviously missing from Mosses' action semantics is *first-class continuations* (noted in [Mos92, p. 211] and [Doh93, §4.1], amongst others). For various reasons, Mosses found continuations difficult and messy to add to action semantics, and so chose to omit them. Unfortunately, this makes a number of language constructs (including some forms of exception handling) exceedingly difficult to specify. Now, with MMAS providing the full power of denotational semantics, continuations can be implemented with relative ease.

Figure 4 shows the definition of the new monad transformer, $ContT$ (some details have been omitted). This monad transformer modifies $return$ and $>>=$ to use continuation-passing style, and defines an operator $callccK$ to perform the call-with-current-continuation operation. By adding this monad transformer to the monad transformers of base MMAS, all existing code will be transparently converted to use a continuation-passing semantics rather than a direct semantics. All existing monad operators are converted to behave appropriately, and the new $callccK$ operator is added.

Recall that at this point, the modularity of the system ensures that (as long as certain proof obligations are satisfied regarding the behaviour of $return_{(ContT\ m)}$, $>>=_{(ContT\ m)}$, and $lift_{(ContT\ m)}$) the behaviour of all existing action notation module code remains identical, and that all proofs of properties of the base MMAS system still remain valid for the new system. *No existing code* need be altered to support the use of this new notion of computation.

In order to use this new feature, however, new action notation must be provided. Figure 5 shows a portion of the code for the new action notation module: the definition of the combinator $cWithCC$ (for which the concrete syntax is with the current continuation in _ do _). In addition to this combinator (and omitted from the figure), two new actions are provided: jump to _ and jump to _ with _. The actions permit a continuation to be invoked, optionally passing it a value; the combinator binds the current continuation to a token in the environment and performs a block of code (the call-with-current-continuation operation, action-semantics style).

The precise semantics of the combinator can be seen by inspecting the modular monadic semantic code in Figure 5. The token yielder is evaluated, and then the $ContT$ operator $callccK$ is used to capture the current continuation. The token is bound to a code fragment that obtains the current transient, passes it to the continuation, obtains the return value and provides it as the transient result. Then the code enclosed by do _ is performed, the resulting transient obtained and returned. Finally, the result of $callccK$ is obtained and returned as a transient to the caller.

With the addition of the monad transformer of Figure 4 and the action notation module of Figure 5 are added, MMAS is extended to include continuations. ASDs can now be written that refer to continuations—see the example in the appendix.

But unlike the other systems depicted in Figure 1, the extension of MMAS has not changed the behaviour or interface of the system with respect to existing features. All existing ASDs will behave just as they did before, and code from them may be used within new ASDs without modifi-

$$
\begin{aligned}
\textbf{type}\ \mathit{ContT\ ans\ m\ a} &= (a \rightarrow m\ ans) \rightarrow m\ ans \\
\mathit{return}_{(ContT\ m)}\ v &= \lambda k \rightarrow k\ v \\
m \gg\!=_{(ContT\ m)} f &= \lambda k \rightarrow m\ (\lambda a \rightarrow f\ a\ k) \\
\mathit{callccK}_{(ContT\ m)}\ f &= \lambda k \rightarrow f\ (\lambda a \rightarrow (\lambda k' \rightarrow k\ a))\ k \\
\mathit{lift}_{(ContT\ m)}\ m &= \lambda k \rightarrow m \gg\!=_m k
\end{aligned}
$$

Figure 4: Continuations: monad transformer

```
cWithCC ytok a =
  ytok >>= λ tok →
  callccK (λk →
    getE >>= λe →
    setE (overlay e
            (bindTo tok
               (abstractAct
                  (getB >>= λ(Tr t) →
                   k t >>= λt' →
                   doB (Tr t'))))) >>= λ() →
    a >>= λ() →
    getB >>= λb →
    return (case b of
              Tr t → t
              _    → tnil))   >>= λt →
  getB >>= λb →
  case b of
    Tr _ → doB (Tr t)
    _    → return ()
```

Figure 5: Continuations: action notation module portion

cation. Even more significantly, the modular proofs that applied to the old ASD and the old version of MMAS will apply equally well to the new extended version! No extra work is required—everything required is encapsulated within the new monad transformer, action notation module and proof obligations.

The full code for the continuation facet of MMAS and for all facets of base MMAS is given in Wansbrough's thesis [Wan97].

6 Related work

The work described here owes a great debt to the work of Mosses and others [Mos92, Mos96a] on action semantics.

Modular monadic semantics derives ultimately from the work of Moggi [Mog89a, Mog89b, Mog91b, Mog91a] on monads for programming language semantics. Steele's work [Ste94] on pseudomonads provided an early prototype of a system very similar to MMAS. Espinosa [Esp95] and Liang, Hudak, and Jones [LHJ95, LH96] provided the details of the implementation of Moggi's ideas in a working system; MMAS is directly based on the work of Liang, Hudak and Jones (the term 'modular monadic semantics' is due to Liang and Hudak [LH96]). It is interesting that both Espinosa and Liang, Hudak, and Jones credit Mosses with inspiring their research. This present paper exhibits a more concrete connection between the two groups.

A number of researchers have developed action interpreters or action compilers [Mou96, Ørb94, BMW92, Pal92], which appear similar to MMAS in that they implement action semantics; but in general these are based on Mosses' structural operational semantics and are constructed monolithically (see Figures 2(a), 2(b) and 2(c)).

Lassen[Las95] and Doh and Schmidt[DS94], like MMAS, replace Mosses' structural operational semantics with an alternative (a reduction semantics and a natural semantics, respectively) and use it to reason about action notation. Neither theory is intended to be executable, however, and neither is particularly modular.

7 Conclusions

In conclusion, then, action semantics is both an excellent system for the specification of domain-specific languages and a fascinating DSL in its own right. However, action semantics has significant limitations. It is incomplete and does not permit extension, and proving results within it is difficult. The solution to this problem is provided by modular monadic action semantics, a tiered system consisting of an action notation layer defined in terms of a modular monadic semantics, which is in turn modularly defined in terms of a functional programming language.

We have seen that modular monadic action semantics enhances action semantics, making it more useful and extending its range of applicability. Through its modularity and extensibility, new features can be added to the base action semantics, enabling the description of languages using these features to be done without resorting to tedious simulation. By replacing Mosses' structural operational semantics with a modular monadic one, we have in fact achieved what Mosses hints at in [Mos96a, §8], where he notes that "the current structural operational semantics of action notation is not so easy to modify; alternative forms ... might be preferable in that respect." We have demonstrated the utility of this by adding continuations to action semantics, in Section 5, something that has been until now quite impractical to achieve.

In addition, the use of a *modular* underlying semantics which is directly based on denotational semantics should make action semantic theory much easier to develop. Proofs will be modular, and can make use of the results and techniques that have been developed in the field of denotational semantics. Of course, as Mosses notes in [Mos96a, §1.3], there are certain technical difficulties with implementing *all* of action semantics in denotational semantics (and hence modular monadic semantics); however MMAS demonstrates that a substantial and useful portion of it *can* be so implemented. As further developments occur in denotational semantics, these may be brought into the MMAS framework and used to increase its scope.

The MMAS system demonstrates the utility of Liang, Hudak, and Jones' modular monadic semantics as a lower-level semantic framework. Our implementation consists of around 1200 lines of code, and so is a significantly-sized example of its use. We found that the system worked extremely well, although our experience did suggest some minor alterations to their approach.

Modular monadic action semantics is a flexible, modular, extensible version of Mosses' action semantics. It allows new features to be readily added to the semantics in a modular fashion, and promises to make the semantic theory more manageable. As such, we believe it offers an excellent extension to action semantics for specifying the semantics of domain-specific languages.

References

[Abr96] Samson Abramsky. Semantics of interaction. In Hélène Kirchner, editor, *Trees in Algebra and Programming—CAAP'96: Proceedings of the 21st International Colloquium, Linköping, Sweden, April 1996*, number 1059 in Lecture Notes in Computer Science, page 1. Springer-Verlag, 1996. Invited talk.

[BMW92] Deryck Brown, Hermano Moura, and David A. Watt. ACTRESS: an action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Compiler Construction: Proceedings of the 4th International Conference, CC'92, Paderborn, FRG, October 1992*, number 641 in Lecture Notes in Computer Science, pages 95–109. Springer-Verlag, 1992.

[Doh93] Kyung-Goo Doh. Action semantics: A tool for developing programming languages. In *Proceedings of InfoScience'93, International Conference on Information Science and Technology*, Seoul, Korea, 21–22 October 1993. Also available as Technical Report 93-1-005, The University of Aizu. Available `ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/Doh93IS/`.

[DS94] Kyung-Goo Doh and David A. Schmidt. The facets of action semantics: Some principles and applications (extended abstract). In Mosses [Mos94], pages 1–15.

[Esp93] David Espinosa. Modular denotational semantics. Unpublished manuscript, December 1993.

[Esp94] David Espinosa. Semantic Lego. Unpublished manuscript, January 1994. Available `http://www-swiss.ai.mit.edu/ftpdir/users/dae/`.

[Esp95] David A. Espinosa. *Semantic Lego*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1995. Available `http://www-swiss.ai.mit.edu/ftpdir/users/dae/`.

[FHK84] Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments: Report on a Workshop directed by F. L. Bauer and H. Remus*, volume 8 of *NATO ASI Series F: Computer and System Sciences*, pages 263–274. Springer-Verlag, 1984.

[HT94] B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In Mosses [Mos94], pages 34–42.

[Las95] Søren B. Lassen. Basic action theory. Technical Report RS-95-25, BRICS, Department of Computer Science, University of Aarhus, May 1995.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP '96: 6th European Symposium on Programming, Linköping, Sweden, April 1996*, number 1058 in Lecture Notes in Computer Science, pages 219–234. Springer-Verlag, 1996.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 22–25, 1995*, pages 333–343, New York, 1995. ACM Press.

[Mog89a] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1989. Available `http://theory.doc.ic.ac.uk:80/tfm/papers/MoggiE/abs-view.ps.gz`.

[Mog89b] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989.

[Mog91a] Eugenio Moggi. A modular approach to denotational semantics. In D. H. Pitt et al., editors, *Proceedings of Category Theory and Computer Science, Paris, France, September 3–6, 1991*, number 530 in Lecture Notes in Computer Science, pages 138–139. Springer-Verlag, 1991. Invited talk.

[Mog91b] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Mos92] Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Mos94] Peter D. Mosses, editor. *Proceedings of the First International Workshop on Action Semantics, Edinburgh, 14 April 1994*, number NS-94-1 in BRICS Notes, 1994.

[Mos96a] Peter D. Mosses. Theory and practice of action semantics. In *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science, Cracow, Poland, September 1996*, number 1113 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Mos96b] Peter D. Mosses. A tutorial on action semantics. Tutorial notes for FME'96: Formal Methods Europe, Oxford, 18–22 March 1996, 1996. Draft. Available `ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/Mosses96DRAFT/`.

[Mou96] Hermano Moura. An implementation of action semantics (draft). Available `http://www.di.ufpe.br/~rat/`, 1996.

[NT94] J. P. Nielsen and J. U. Toft. Formal specification of ANDF, existing subset. Technical Report 202104/RPT/19, issue 2, DDC International A/S, Lundtoftevej 1C, DK-2800 Lyngby, Denmark, 1994.

[Ørb94] Peter Ørbæk. OASIS: An optimizing action-based compiler generator. In Peter A. Fritzson, editor, *Compiler Construction: Proceedings of the 5th International Conference, CC'94*, volume 786 of *Lecture Notes in Computer Science*, pages 1–15, Edinburgh, U.K., April 1994. Springer-Verlag.

[Pal92] Jens Palsberg. A provably correct compiler generator. In B. Krieg-Brückner, editor, *ESOP '92: Proceedings of the 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434, Rennes, France, February 1992. Springer-Verlag.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Department of Computer Science, University of Aarhus, 1981. Now available only from University of Edinburgh.

[Plo83] G. D. Plotkin. An operational semantics for CSP. In Dines Bjørner, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts—II, Garmisch-Partenkirchen, FRG, 1–4 June 1982*, pages 199–223, Amsterdam, 1983. North-Holland.

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, 1994. ACM Press.

[vDM96] Arie van Deursen and Peter D. Mosses. *ASD: The Action Semantic Description Tools User's Guide*, May 24 1996. Version 2.5. Available `http://www.brics.dk/Projects/AS/Tools/ASD/`.

[Wan97] Keith Wansbrough. A modular monadic action semantics. Master's thesis, Department of Computer Science, University of Auckland, February 1997. Available `http://www.cs.auckland.ac.nz/~kwan001/thesis/`

A Devils and Angels

This is a problem from [FHK84]. Carlsson uses it as an example in his Advanced Functional Programming course at Chalmers. His solution code (in Haskell, using a primitive version of [LHJ95]'s monad transformer system) can be found at `http://www.cs.chalmers.se/~magnus/afp/problems/devils-n-angels/`; this code follows Friedman, Haynes and Kohlbecker's Scheme code in [FHK84].

The problem is to define three actions, milestone, devil and angel, with the following behaviour. The computation has the goal of finishing despite the existence of devils.

Whenever a devil is encountered, control is sent back to the last milestone. If another devil (or the same one again) is encountered, control is sent back to the milestone before that one, and so on. If no milestones remain, the devil does nothing.

Whenever an angel is encountered, control is sent forward to where the computation last met a devil. If another angel is encountered, control is sent further forward to the devil before that one, and so on. Again, if no devils have been encountered the angel does nothing.

The milestone appears as an action that simply passes a transient straight through, like regive. The value passed to a devil, however, is given to what follows the appropriate milestone; the value passed to an angel is given to what follows the appropriate devil.

Continuations provide an excellent means of implementing the above problem. We maintain two stacks of continuations: one of *past* continuations (pushed by milestones and popped by devils), and one of *future* continuations (pushed by devils and popped by angels). This is achieved by the following actions:

```
(1)  pop-cont S:Token =
         give the data stored in
              the cell bound to S then
         | | check the count of it
         | |         is greater than 0 and then
         | | | | give the first of it
         | | | and
         | | | | store the rest of it in
         | | | |        the cell bound to S
         | or
         | | check the count of it is equal to 0 then
         | | | give the abstraction of regive .

(2)  push-cont ( S:Token, Y:Yielder ) =
         | | give Y
         | and
         | | give the data stored in
         | |      the cell bound to S
         then
         | store it in the cell bound to S .
```

Note that the continuations are stored in a tuple stored in a named cell. We use cells named "past" and "future".

We can now define the required primitives, milestone, devil and angel, as follows:

```
(3)  milestone =
         with the current continuation in "k" do
         | | push-cont ( "past", the data bound to "k" )
         | and
         | | regive .

(4)  devil =
         with the current continuation in "k" do
         | | push-cont ( "future", the data bound to "k" )
         | and
         | | | pop-cont "past" and regive
         | | then
         | | | jump to the given continuation#1
         | | |        with the given datum#2 .

(5)  angel =
         | pop-cont "future" and regive
         then
         | jump to the given continuation#1
         |        with the given datum#2 .
```

milestone simply obtains the current continuation and pushes it onto the "past" stack, and then passes through the value passed to it.

devil obtains the current continuation, pushes it onto the "future" stack, and then passes the value passed it to the continuation popped off the top of the "past" stack (note that if the stack is empty, we are given the identity abstraction abstraction of regive, so we get the correct behaviour even in this case).

angel simply passes the value it is passed directly to the continuation popped from the top of the "future" stack. Again, if it is empty it uses the identity abstraction.

We conclude with code for a short example due to Carlsson:

```
(6)  supernatural =
         | allocate a cell then bind "past" to it moreover
         | | allocate a cell then bind "future" to it
         hence
         | | store () in the cell bound to "past" and
         | | | store () in the cell bound to "future"
         | then
         | | give 1 then
         | | | milestone then
         | | | | | check it is equal to 1 then
         | | | | | | give 2 then
         | | | | | | | devil then
         | | | | | | | | give the sum of ( it, 100 )
         | | | | or
         | | | | | check it is not equal to 1 and then
         | | | | | | give the sum of ( 3, it ) then
         | | | | | | | angel .
```

After allocating cells for the two stacks and initialising them, this code passes 1 to the first milestone; if it returns 1 then it passes 2 to a devil and gives a final result of whatever the devil returns plus 100. If the milestone doesn't return 1 then the code adds 3 to whatever the milestone did return and passes the result to an angel.

Execution proceeds as follows: 1 is passed to the milestone, and 1 is returned. 2 is passed to the devil, which jumps back to the milestone and returns 2 from it. 3 is added to 2 to get 5, which is passed to the angel. The angel jumps forward to the devil, which now returns 5; 100 is added to this to get 105, which is returned from the computation as the final result.

SHIFT and SMART-AHS: A Language For Hybrid System Engineering Modeling and Simulation

Marco Antoniotti Aleks Göllü
California PATH
University of California at Berkeley
{marcoxa,gollu}@path.berkeley.edu

Abstract

SHIFT *is a new programming language, whose aim is to facilitate the implementation of* reusable simulation frameworks *by teams of engineers.* SHIFT *incorporates system theoretic concepts emerging from the field of* Hybrid Systems *analysis and modeling. The SMART AHS framework is a collection of* SHIFT *libraries devoted to the construction of* Hybrid System *based simulation of* Automated Highways Systems. *In this paper we describe how the* SHIFT *simulation environment and language have impacted on the development of the SMART AHS framework. Our claim is that* SHIFT *provides the proper level of abstraction for engineers who face complex modeling and simulation tasks, where phase changes and continuous variables interact in subtle ways.*

1 Introduction

SHIFT is a new programming language, whose aim is to facilitate the implementation of *reusable simulation frameworks* by teams of engineers. SHIFT incorporates system theoretic concepts emerging from the field of *Hybrid Systems* analysis and modeling (e.g. see [1]) into an object-oriented language environment that offers the proper level of abstraction for describing complex applications such as automated highway systems, air traffic control systems, robotic systems, shop floors, coordinated submarines and other systems whose operation cannot be captured easily by conventional models. These application domains share the following key characteristics.

- The behavior of objects in the system have both continuous and discrete event components;
- The systems consist of heterogeneous set of interacting objects where models of individual objects are known and the goal is the study of the emergent behavior resulting from their interaction; and
- A static block diagram representation is not sufficient to specify all data dependencies among objects since the sets interacting objects vary over time.

Moreover the practitioners involved in these projects are, most of the time, engineers who like to work with their own "tools of the trade". SHIFT offers them with two of them – automata and differential equations – well integrated into a single environment and language.

Our work on simulation frameworks was driven by application needs in highway automation. In the early 90s, PATH (Partners for Advanced Transit and Highways) proposed a specific control hierarchy for the highway automation project. The need for a simulation environment was obvious, since we were dealing with a complex and very large system, which gave no hope for a closed form mathematical analysis. Following an unsuccessful market study of the available tools, a decision was made to internally develop a system to simulate the PATH control architecture for highway automation. A C based simulation system, SmartPath, was created [8]. The project evolved and gained national attention.

Following the original proposal, the National Automated Highway System Consortium (NAHSC) was funded and other highway automation architectures were developed. It became evident that a generalized simulation framework was needed that could facilitate the specification, simulation, and evaluation of different highway automation architectures.

As a result SMART AHS$_{C++}$, a C++ based persistent simulation framework was developed [8].

SMART AHS$_{C++}$ used a set of class libraries developed in C++, SmartDB, as its object model, and delivered a framework for further customization by application developers. However, SMART AHS$_{C++}$ had it shortcomings. It did not allow its users to program in their "own" domain which was differential equations and state machine representations. It introduced artificial specification rules and syntax resulting from the use of a general-purpose programming language. We feel that these shortcomings are shared by most simulation libraries embedded in a host language which does not support language extensions (like Simula, C or C++ or Java).

In parallel to our AHS work, we were involved with several other projects, such as air traffic management, power transmission and distribution systems, and network management systems. In system engineering, we have observed a general shift towards hierarchical control of large systems that combined classical continuous feedback systems, with more recent discrete event based control algorithms and protocol specifications. This hybrid systems paradigm has proven ideal for the specification, control, and verification of such complex, large, dynamical systems.

Our experience with a multitude of such systems resulted in a set of requirements for frameworks for the design, specification, control, simulation, and evaluation of large dynamical systems. No language, product, or tool in the market nor in academia satisfied all requirements.

The design and implementation of a language that addressed all the requirements required expertise from several disciplines including computer science, electrical engineering, and mechanical engineering. Such a multi-disciplinary team was assembled at PATH/UC Berkeley and a new programming language, SHIFT, was born.

This paper provides an overview of the concepts and constructs of the SHIFT language, and a discussion of the impact on the development cycle of simulation model illustrated through the SMART AHS case study. The SHIFT mathematical model is beyond the scope of this paper: we refer the interested reader to [7] for a comprehensive exposition.

1.1 Related Work

SHIFT is used to describe models with switched differential equations (such as a vehicle with automatic gear shift) and coordinated behaviors (such as communicating controllers). Standard math and simulation tools such as Matlab^{tm}, Mathematica^{tm}, Maple^{tm} or Matrix$_X$^{tm}, while suitable for numerical or symbolic integration of fixed sets of differential equations, are difficult to use in applications with rapidly changing sets of differential equations (due to the evolution of relationships among components), complex event-triggering conditions (such as existential queries on the state of the world), and complex program logic (such as synchronous compositions of state machines). More traditional *discrete event* simulation packages like GPSS^{tm}, while offering a tried and tested base, lack the facilities for writing concise hybrid systems models.

The hybrid systems approach [1] satisfies our needs for component modeling but it lacks the capacity to model dynamically reconfigurable interactions between components.

The Omola/Omsim [11] language has a very similar approach to hybrid system modeling as SHIFT. Both systems provide a modeling language with simulation semantics; both support discrete event and continuous time behavior representation; both have the necessary constructs for hierarchical modeling and specification reuse. However, Omola is designed to represent statically interconnected objects. Furthermore, it does not provide the means to manipulate sets and arrays of components. In SHIFT, these manipulations are used to express and compute the evolution of the interconnections among components as the world evolves.

Statecharts [9] and Argos [10], based on Statecharts, are approaches for synchronous discrete event modeling. Their focus is on hierarchical specification of finite state machines. SHIFT does not provide explicit facilities for hierarchical behavior specification; instead, it provides a sub-typing mechanism wherein a subtype (presumably more detailed) must present the same interface as its super-type. SHIFT adds continuous time semantics and dynamic reconfigurability of the synchronization structure. Subtyping and other constructs may be used to organize components hierarchically.

Recent extensions to the DEVS [16] formalism have introduced notions of dynamic reconfiguration [4, 13]. However, the DEVS formalism is primarily aimed at discrete event simulation and the extensions for continuous evolution laws are limited. Model specification in DEVS is done with C++, SmallTalk or Common Lisp classes that implement the mathematical model at hand, requiring the user to work at the host language level.

1.2 Preliminary Discussion

Our mathematical model for the discrete event semantics is similar to Milner's π-calculus [12]. Both models achieve reconfiguration by a renaming of event labels used in synchronization. The finite state machine part of SHIFT implements this model. The differential equation part of SHIFT allows systems of first order ODE's.

The abstraction facilities in general-purpose programming languages such as the original Simula or C/C++, although powerful enough to encode our models, would not allow us to write simple, concise descriptions of our designs. The best that could be hoped for, would be an integration at the level of "embedded interpreters" *à la* Tcl/Tk or SQL[1].

SHIFT provides both high-level system abstractions and the flexibility of a programming language. However, all the features that SHIFT offers are carefully designed to constrain the programming style and to conform to the underlying mathematical model, while avoiding frustration for the user.

As a first statement about the impact of SHIFT in the programming of complex simulations, when we will discuss the reimplementation of SMART AHS in SHIFT in Section 3, we will see that the size of the resulting "libraries" and "projects" decreased by almost 50%, while the code could be more easily reused.

Users of SHIFT within the PATH project, NAHSC and UCB reported favourably on the ease with which their engineering models were readily translated into working simulations.

Moreover, since a SHIFT program is a direct implementation of a hybrid system specification (even though an extended one), the resulting code can be easily manipulated and fed into the new breed of *automated verification systems* like KRONOS [6].

Though not substantiated by a direct comparative study, but only by an "a posteriori" examination of the evolution of SHIFT, SmartPATH, SMART $\text{AHS}_{\text{C++}}$ and SMART AHS, we claim that these results justify the considerable research and implementation effort that went into the develoment of these new tools.

[1]The SHIFT systems provides a C API for this style of programming.

2 SHIFT Language Overview

A SHIFT program describes a set of interacting objects called *components* and grouped into *component types*[2]. The SHIFT *type declaration* construct specifies the prototypical behavior of all components of a given type. SHIFT supports a single inheritance scheme which has proven sufficient for our needs.

The set of components types and their instances in a SHIFT program, directly describe a *hybrid system* with comprises synchronizing finite states machines and differential equations.

SHIFT additionally supports a small set of basic data types (number and symbol) – and of constructed types (array and set). The set of built-in types has the following characteristics:

- Objects of type number have piecewise constant or piecewise continuous real-valued time traces. The latter variables have type continuous number.
- Objects of type symbol have piecewise constant symbol-valued time traces. In SHIFT symbols are similar to C enumeration tags. However they do not require a declaration.
- An object of type set(T), where T is a native or user-defined type, contains a set of elements of type T.
- An object of type array(T) contains a one-dimensional array of elements of type T, whose dimension is determined at creation time.

A component prototype is defined by the SHIFT type declaration. The structure of a type roughly consists of

- *inputs*
  instance variables (or simply "variables") which can be read but not changed by the behavior of the component and which are visible outside the scope of the type definition.
- *outputs*
  instance variables which can be read and changed by the behavior of the component and

[2]Our terminology abuses words like *type*. Using more standard Object Oriented terminology, we would speak of *instances* and *classes*. We use the term "component" since the control theory application domain imposes a natural *part-of* metaphor on the software architecture.

which are visible outside the scope of the type definition.

- *states*
  instance variables which can be read and changed by the behavior of the component but that are not visible outside the scope of the type definition.

- *discrete modes and transitions*
  i.e. the definition of the finite state machine behavior of the type.

- *differential and algebraic equations*
  i.e. the definition of the continuous behavior of the type.

The terminology is taken from the standard Control Theory practice and it roughly translates into the well know concepts of *private* and *public* slots in a class. Also, the notions of *inputs* and *outputs* are supported by the language in order to promote a "black box" software development style.

As an example, here is a first SHIFT code fragment:

```
type car
{
        input
                continuous number throttle;
        output
                continuous number position, velocity;
                continuous number acceleration;
        state
                continuous number fuel_level;
                car car_in_front;
                controller controller;
                ...
}
```

The *discrete finite state* behavior and the *continuous behavior* of a type are specified in different "clauses" of a user definition.

The continuous behavior is specified by ordinary differential equations and algebraic definitions which are grouped under the flow clause. Each instance variable can be used in these equations and their behavior is computed accordingly[3]. Each equation group (appropriately called a *flow*) can be labeled with a meaningful name. The default flow contains the equations which are to be used whenever there are no special provision for computing the value of the variables involved.

[3]Of course, only *continuous* number variables make sense in a differential definition.

The discrete clause defines the possible values for the type's mode (i.e. the finite state "current state") and associates to each of them a set of differential equations and algebraic definitions or one of the flows defined in the flow clause.

The differential equations are specified by systems of first order ODE of the form $x' = f(x, u)$, where x is a single variable and u is a vector of "other" variables. The algebraic definitions cannot contain circular dependencies. Such dependencies are detected at run-time, and an error is signaled by the run-time system.

As an example (continuing the "car" example):

```
type car
{
        ...
        flow
                default {
                        position' = velocity;
                        velocity' = acceleration;
                }
        discrete
                accelerating      { acceleration = 3; },
                cruising          { velocity = 30; },
                brake             { acceleration = -5; };
}
```

Notice that the *cruising* state redefines *velocity*, which becomes algebraically defined (as a constant in this case) instead of differentially defined (as the integral of the acceleration).

Transitions between discrete modes are defined in the transition clause, as in the following example.

```
type car
{
        ...
        transition
                accelerating -> cruising {}
                when velocity >= 30,
                cruising -> braking {}
                when position(car_in_front) - position < 5;
}
```

The example uses the state variable *car_in_front* containing a reference to another *car*, whose relative position is used in deciding when to apply brakes.

Transitions are labeled by a (possibly empty) set of event labels. These labels allow transitions to synchronize with each other. Moreover, transitions may be *guarded* by boolean expressions – introduced by the when keyword – and may trigger a set of actions grouped in a do clause. These actions *reset*[4] (i.e. assign) the values of variables, may create new components and may reconnect their inputs and outputs.

Suppose that we wish the car to brake when a roadside controller signals an emergency. This can be specified with the transition

```
type car
{
    ...
        cruising -> braking {controller:emergency}
        when position(car_in_front) - position < 5;
}
```

The definition of the *controller* type includes an *exported event*, *emergency*, and a transition that triggers it.

```
type controller
{
    export emergency, ...;
    discrete normal, panic_mode, ...;
    transition
        normal -> panic_mode {emergency}
        when some critical condition;
    ...
}
```

SHIFT allows the system modeler to specify very complex patterns of *synchronous composition* of finite state automata. The transition guards may contain existential quantifiers that query the state of sets of components (possibly all existing components). For example, let *cars* be the set of all the components of the *car* type and let the road consist of a single lane. Then, the variable *car_in_front* is updated as follows.

```
type car
{
    ...
    transition
        cruising -> cruising {}
        when exists c in cars :
            position(c) > position
            and position(car_in_front) > position(c)
        do {
            car_in_front := c;
        },
        ...
}
```

[4]The terminology, once again is borrowed from the field of hybrid system studies.

The initializations of a newly-created component of some type are defined in the setup clause. For example, each component of type *car* may add itself to the set *cars* when it is created.

```
type car
{
    ...
    setup
        do {
            cars := cars + { self };
        };
}
```

In practice a SHIFT program would not use this exact code unless *cars* were a small set. A more efficient mechanism requires maintaining multiple sets of cars associated with lanes and highway segments.

2.1 SHIFT Support Environment

SHIFT has many more features which we do not discuss here in further detail since they are outside the scope of the paper[5]. We now briefly comment on the SHIFT support environment and implementation.

SHIFT programs are translated directly into C by the `shic` compiler. The resulting C file is then linked with the SHIFT runtime library in order to produce an executable (a file conventionally ending with `.sim`). The runtime library takes care of the implementation of the high level data structures used by SHIFT (e.g. sets) and makes provisions to integrate the differential equations via standard Runge-Kutta algorithms. There are no special optimizations that are done by the compiler[6]. The only requirement imposed on the system is that the behavior of the run-time which essentially interprets the set of finite state machines and differential equations complies with underlying mathematical model.

[5]Among them: single inheritance, complex set and array formers à la SETL [14], garbage collection (using Boehm's conservative GC [5]), a foreign function interface facility and a C API which allows an experienced programmer to control SHIFT simulations from C and C++ programs.

[6]As a matter of fact, many constructs and compilation policies could be optimized away by some rather simple data flow analysis. However, this has not been so far the emphasis of our work.

There are several subtleties involved in the interaction between the RK integration routine and the guard evaluation code. Languages like Omola/Omsim that support only a static set of differential equations can perform compile-time optimizations to select integration step sizes. In SHIFT since the dependencies among the components can change at run-time, it is not possible to optimize integration step-sizes with respect to guard-crossings. SHIFT applications so far have been primarily in non-stiff systems, hence a fixed-step RK integration algorithm had the best performance. We recognize that this is still an open research field.

The executable file can be run is two ways: by starting a command line monitor or by connecting the simulation with a Tcl/Tk graphical user interface in a client/server fashion[7] (see Figure 1 for a screen shot).

Both the command line monitor and the graphical environment allow the user to control the running simulation. Typical operations include

- data inspection
- stepping by time click and by "simulated time"
- stopping and resumption of execution in correspondence of discrete transitions.

These operations support the simulation modeler "at the right level" of abstraction and allow her/him to quickly determine whether there are logical problems in the code.

3 Developing Simulation Frameworks in SHIFT: The SMART AHS Case

We used SHIFT to develop a specialized framework (SMART AHS) for the construction of simulation models of highways. The overall design principles were first described in [15]. The objectives of the SMART AHS framework are listed hereafter.

[7]The choice to use a client server architecture for the GUI was based on two considerations. (1) The development of the GUI could progress rather independently of the development on the SHIFT compiler and runtime. (2) It was recognized since the early design stages that SHIFT simulations could require huge memory spaces: hence the necessity to being able to run simulation remotely on powerful workstation while interacting with them on a local – and less powerful – machine.

1. To provide researchers with a standardized tool which can be used for evaluation of simulation results under different policies.
2. To allow the quick construction of alternative simulation models.
3. To allow the simulation of models at different granularity levels.
4. To be able to handle medium to large scale simulations.

In Sections 3.1 and 3.2 we describe the SMART AHS architecture and discuss the lessons learned in its deployment as one of the standard tools used by the National Automated Highway System Consortium (NAHSC).

3.1 SMART AHS Architecture

The SMART AHS framework is roughly divided into two parts. The first is a "static" part which contains highway type definitions used to compose different highway layouts. The second part contains different vehicle models used for diverse simulations.

The *highway* types comprise *Lane*, *Section*, *Segment*, *Barrier*, *Block* and *Weather*. These types and their structure constitute a *data description language* for highways.

The *vehicle* types are centered around a container type called *AutomatedVehicle*. Its most immediate sub-components are *Controller*, *VehicleModel* and *VREP* (*Vehicle Roadway Environment Processor*). Figure 2 contains a schematic representation of the *AutomatedVehicle* SHIFT code.

This architecture meets the requirements by allowing the system modeler to plug in different controllers and vehicle dynamic models. Researchers at PATH have successfully developed two classes of models for highly detailed vehicle dynamics simulation and for high volume highway simulations with complex vehicle maneuvers.

The detailed simulation model describes a vehicle at the level of gear shifting and engine dynamics. The model is realistic and based on real data collected by General Motor researchers.

The "high volume" simulation model uses a simplified vehicle dynamics model (there is no need to simulate the engine dynamics when computing flows over stretches of highway) and a controller

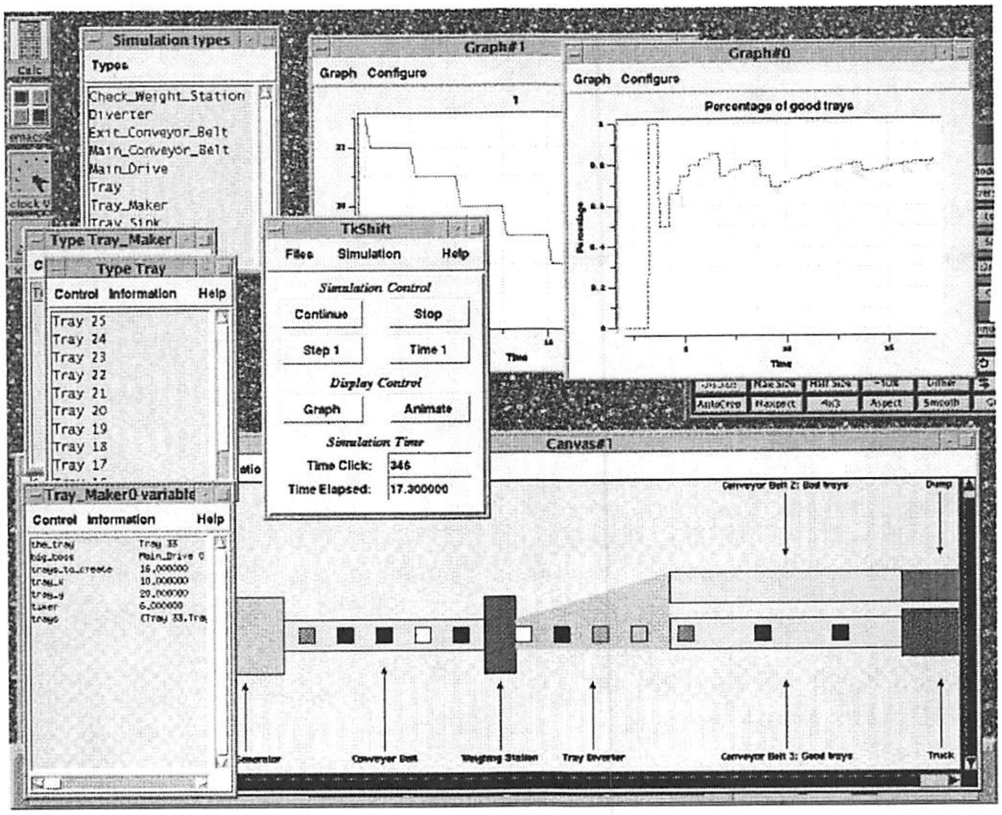


Figure 1: A screen shot of the Tcl/Tk SHIFT environment. The simulation being run models a food processing manufacturing line.

which is devoted to maintain *safety* parameters (e.g. distance from the vehicle in front) and to perform *merge* maneuvers on entrance and exit ramps.

3.1.1 Micro Simulation of Houston Metro Katy Corridor

The "high volume model" was developed to gather data for a project sponsored by the Houston Metro Transportation Authority. The project asked evaluation data for a preliminary design of a stretch of highway with three entry ramps and three exit ramps. The main objective of the study was to evaluate the congestion build up at the three entry ramps under different demand volumes with *autonomous* and highly automated vehicles[8]. Other parameters which were under study include the required length of the entry ramps to ensure completion of merge maneuvers. A more detailed description of the experiment and of its results is contained in [2, 3].

Most of the work done to develop the simulation went into the construction of a *Controller* module which would obey a distributed protocol for cruising and merging into existing traffic. A code fragment for the *Controller* is shown in Figure 3.

The GUI environment was used to visualize the results of the simulation in order to spot problematic areas of the protocol.

We tested four cases on a matrix given by *high* and *low* traffic demand and by enforcing two different *vehicle tracking* policies. The traffic demands levels can be summarized as follows:

low ca. 2000 vehicles/hour injected in the highway system.

high ca. 4000 vehicles/hour injected in the highway system.

In each case each of the vehicles was simulated by instantiating a full blown *AutomatedVechicle* con-

[8]The term "autonomous" is here intended in the following sense: a vehicle/driver which takes decisions based only on its sensor input and on certain assumptions on the behavior of nearby vehicles. The high automation characteristic can be thought of as modern *Adaptive Cruise Control* technology.

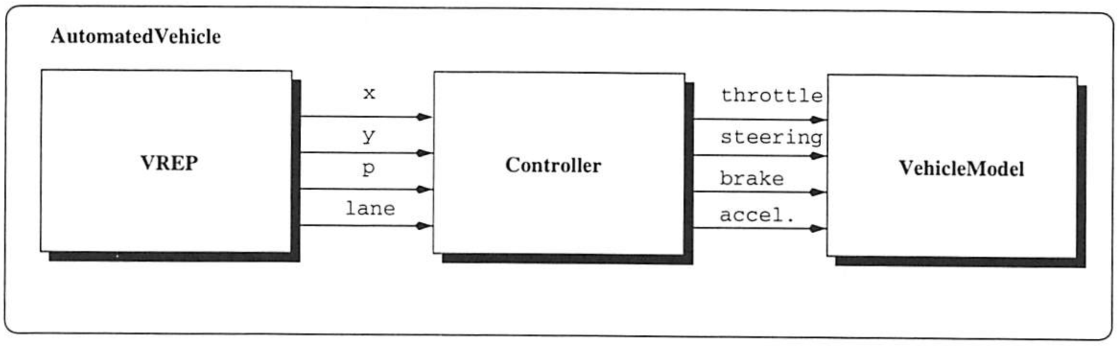


Figure 2: Schematic Block Diagram of *AutomatedVehicle* SHIFT code.

tainer, and each instance obeyed the protocol implemented in the *Controller*. The overall speed of the simulation, depending on the size of the input data was always between 4 and 9 times of *simulated physical time*[9].

3.2 Discussion and Evaluation of the SMART AHS Framework

The construction of the SMART AHS framework and of applications based on it, has given us an insight on how to conduct the development of an harmonious set of libraries. This was a requirement of the overall project and our aim. We consider our experience so far successful and we credit this success to two main characteristics of the SHIFT and SMART AHS framework.

- SHIFT provides a sound and restricted mathematical model (Hybrid Systems) which can be successfully mastered by an engineer in a rather short period of time. The tools provided (differential equations and finite state automata) are the "right" ones for the kind of models we were targeting.

- SHIFT and SMART AHS code is significantly more compact. The SHIFT implementation of SMART AHS consists of 3,300 lines of SHIFT code and 2,000 lines of C legacy code[10], plus about 5,000 lines for the full SHIFT runtime[11]. The Houston-Metro case study consisted of 3,800 additional lines of SHIFT code. The predecessor of SMART AHS (SMART $\text{AHS}_{\text{C++}}$) consisted over 20,000 lines of C++ code, without the code for the distributed merging controller and the highway building

- The SMART AHS framework has a very small set of "how-to-use" rules that a programmer needs to know about. As a result new users can immediately become more productive in application development. Based on our experience with the previous incarnations of SMART AHS, frameworks based on C/C++ usually have too many "how-to-use" rules that are not enforced by any compiler and result in unpredictable run-time errors if not followed properly.

Of course these remarks can be taken as a simple recipe for "good engineering practice", yet we claim that the overall design of the tools did pay off considerably.

The entire simulation study of the Houston Katy Corridor was built from scratch (i.e. the highway layout, the vehicle dynamics models, the merge and cruise protocol and the actual simulation runs) in less that three work weeks.

The merge simulation case study is being followed by more complex studies regarding

- emissions evaluation,

- coordination protocols involving radio communications,

- "platooning" of cars on automated highways, and

- detailed physical simulation of crash and "near-miss" situations for safety analysis.

[9] However, the slow down is due mainly to the current implementation of sets in SHIFT. New tests will be performed with a new implementation which will improve on the memory usage of the internal data structures.

[10] Mostly, I/O routines for the uploading of the engine model data.

[11] Which includes all the necessary development hooks, GUI hooks, C API and Foreign Function Interface.

```
type Controller
{
    output
        continuous number acceleration;
    ...
    state
        Vehicle the_vehicle, side_lane_vehicle;
        continuous number same_lane_accel;
        number nominal_speed;
    ...
    discrete
        cruise{cruise_law};
        yield {yield_law};
    ...
    flow
        default {
            same_lane_accel =
                track_acceleration(same_lane_rel_speed, xDot(the_vehicle), nominal_speed);
        };
        cruise_law {
            acceleration = same_lane_accel;
        };
        yield_law {
            acceleration = min(same_lane_accel, side_lane_accel)
        };
    ...
    transition
        cruise -> yield
            when rear_gxp(the_vehicle) >= L_gap_visible_range(junction)
                and exists mp in merging_vehicles(junction) :
                    ((rear_gxp(mp) >= front_gxp(the_vehicle))
                    and (rear_gxp(mp) <= front_gxp(the_vehicle) + lateral_sensor_range)
                    and (xDot(the_vehicle) <= xDot(mp) + yield_rel_speed_threshold))
            do {
                side_lane_vehicle := mp;
            },
    ...
}
```

Figure 3: A fragment of the *Controller* code. The fragment shows the transition that takes the (instance of) the *Controller* of *the_vehicle* from the *cruise* to the *yield* state. The condition upon which this transition is allowed is expressed in the when clause. The variables suffixed by *_gxp* and *_gyp* represent positions. The accessor *xDot(the_vehicle)* is integrated to the current speed of the vehicle.
When the transition has taken place, the integration will use a modified set of equations to produce the value for the *acceleration* parameter, whose computation eventually relies on a C function (*track_acceleration*).

These projects simply extend the framework or change the simulation granularity, confirming our claim that the level of abstraction provided by SHIFT and SMART AHS is the proper one. Other SHIFT applications developed within UCB Mechanical Engineering and Electrical Engineering departments also confirm that the model/simulate/analyze cycle improves considerably when compared to more traditional approaches applied to similar problems.

3.2.1 Preliminary Cost Analysis

The overall SHIFT design and development took about 18 months for a core group, averaging six people (though the complete list of people who actually contributed is much longer). The first version of SHIFT became available in September of 1996 and it did not include many of the features that were introduced later, during the winter of 1996/97. Currently there are five projects directly funded by either PATH or the NAHSC that are using SHIFT and SMART AHS. These projects directly involve about 20 people for the development and the interpretation of the simulation results. New projects will be added to this list as the FY 98, as the NAHSC expands and redirects its efforts.

The cost for the Houston case study turned out to be in the order of 3 men/month. Subsequent projects (emission control simulation, platooning and coordination) reused much of the overall structure developed in the first place for the Houston Case Study and showed a faster turnaround of the simulation results.

SHIFT is used in other application domains such as autonomous underwater vehicles, and air traffic management simulations. However, our group has not undertaken a formal project tracking effort in order to evaluate the overall impact of the technology outside California PATH.

4 Conclusion

In this paper we presented SHIFT: a new programming language based on theoretic concepts emerging from the field of *hybrid systems*. We have claimed that SHIFT offers the proper level of abstraction for describing complex applications such as automated highway systems, air traffic control systems, robotic shop floors, coordinated submarines and other systems whose operation cannot be captured easily by conventional models.

To support our claim we have described our experience with the SMART AHS framework for the simulation of complex highway systems. Our experience indicates that SHIFT and SMART AHS do achieve the objectives that were at the base of its design.

In particular, SHIFT is currently enjoying a growing popularity and is being used as a teaching tool in various courses in the Electrical Engineer Department of UC Berkeley.

Future work on SHIFT will include the following items:

- further research on the interaction between the integration and guard crossing algorithms;
- parallelization and distribution of the run-time system;
- integration with automated verification systems such as KRONOS [6].

As already mentioned, at this point we cannot provide a direct comparative study of the "simulation development costs" for SHIFT and SMART AHS with respect to a more traditional approach based on standardized libraries. Setting up such a study would require a considerable effort in itself and the identification of a proper set of tools to compare SHIFT and SMART AHS against. However, the feedback we gathered from the users of SHIFT makes us very confident that the results would tip the balance in its direction.

5 Acknowledgments

We wish to thank all the people at California PATH and elsewhere, especially A. Deshpande, F. Eskafi, A. Girault, M. Kourjanski, J. Misener, V. Murgier, L. Semenzato, J. Sousa, P. Varaiya, D. Weismann, S. Yovine, and the National Automated Highway System Consortium

6 Availability

Of course, SHIFT and SMART AHS can be downloaded for free under a UCB-style license from our home pages

- `http://www.path.berkeley.edu/`
- `http://www.path.berkeley.edu/shift`
- `http://www.path.berkeley.edu/smart-ahs`

and our ftp site

- `ftp.path.berkeley.edu:pub/PATH/SHIFT`
- `ftp.path.berkeley.edu:pub/PATH/SMART-AHS`

References

[1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In R. L. Grossman, A. Nerode, A. P.

Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[2] M. Antoniotti, A. Deshpande, and A. Girault. Microsimulation analysis of automated vehicles on multiple merge junction highways. In *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics (SMC97)*. IEEE, October 1997.

[3] M. Antoniotti, A. Deshpande, and A. Girault. Microsimulation analysis of multiple merge junctions under autonomous ahs operation. In *Proceedings of the IEEE Conference on Intelligent Transportation Systems (ITSC97)*. IEEE, November 1997.

[4] F. Barros. Dynamic Structure Discrete Event System Specification Formalism. *Transactions of the Society for Computer Simulation*, 1:35–46, 1996.

[5] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, pages 807–820, September 1988.

[6] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[7] A. Deshpande, A. Göllü, and P. Varaiya. A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata. In *Hybrid Systems IV*. Springer-Verlag, 1997.

[8] F. Eskafi, D. Khorramabadi, and P. Varaiya. An Automated Highway System Simulator. *Transportation Research Journal, part C*, 3(1), 1995.

[9] D. Harel. Statecharts: A Visual Approach To Complex Systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[10] F. Maraninchi. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In *Proceedings of the IEEE International Conference on Visual Languages*. IEEE, 1991.

[11] S. E. Mattson and M. Anderson. The Ideas Behind Omola. In *Proceedings of the IEEE Symposium on Computer Aided Control System Design, CADCS 1992*. IEEE, March 1992.

[12] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.

[13] H. Praehofer, F. Auernig, and G. Resinger. An Environment for DEVS-based Multiformalisms Simulation in Common Lisp/CLOS. *Discrete Event Dynamic Systems: Theory and Application*, 3(2):119–149, 1993.

[14] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets. An Introduction to SETL*. Springer-Verlag, 1986.

[15] P. Varaiya. Smart Cars on Smart Roads: Problems of Control. *IEEE Transactions on Automatic Control*, 38(2):195–207, February 1993.

[16] B. Zeigler. *Multifaceted Modeling and Discrete Event Simulation*. Academic Press, London, Orlando, 1984.

Design and Semantics of *Q*uantum: a Language to Control Resource Consumption in Distributed Computing.

Luc Moreau
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
L.Moreau@ecs.soton.ac.uk

Christian Queinnec
LIP 6 & INRIA-Rocquencourt
4 place Jussieu, 75252 Paris Cedex
Christian.Queinnec@inria.fr

Abstract

This paper describes the semantics of *Q*uantum, a language that was specifically designed to control resource consumption of distributed computations, such as mobile agent style applications. In *Q*uantum, computations can be driven by mastering their resource consumption. Resources can be understood as processors cycles, geographical expansion, bandwidth or duration of communications, etc. We adopt a generic view by saying that computations need energy to be performed. Quantum relies on three new primitives that deal with energy. The first primitive creates a tank of energy associated with a computation. Asynchronous notifications inform the user of energy exhaustion and computation termination. The other two primitives allow us to implement suspension and resumption of computations by emptying a tank and by supplying more energy to a tank. The semantics takes the form of an abstract machine with explicit parallelism and energy-related primitives.

1 Introduction

Millions of computers are now connected by the Internet. At a fast pace, applications are taking advantage of these new capabilities, and are becoming parallel and distributed. For instance, mobile agents [Mag96a] and multi-agent systems are technologies used in a wide range of activities, such as information discovery on the WWW [DD97] or negotiation on behalf of the user [WJ95]; they exploit parallelism to improve efficiency and autonomy, and they rely on distribution or mobility to increase locality.
A major challenge is to be able to control and monitor computations in such a distributed context. On the one hand, users are ready to delegate negotiation power and responsibility to their agents, but they wish to bound their activities by geographical, temporal, physical (such as memory, ...), or monetary constraints; furthermore, during execution, users might wish to dynamically monitor and control their agent's behaviour by reducing or adding constraints. On the other hand, service providers offer platforms where mobile agents can migrate to in order to use available facilities; they are anxious to ensure that visiting agents act according to a previously negotiated agreement, that they do not exceed temporal or physical limits, and that they are charged according to their usage of facilities.
Our goal is to provide the means by which everybody, users and service providers, can control and monitor resources used by parallel and distributed computations. We believe that parallel computations can be driven by mastering their *resource consumption*. Resources can be understood as processor cycles, bandwidth and duration of communications, or even printer paper. We adopt a more generic view by saying that computations need *energy* to be performed[1].
In this paper, we present a new language, called *Q*uantum, that is specifically designed to monitor and control the resource consumption of computations. The essence of *Q*uantum is summarised by three key ideas. *(i)* Quotas of energy can be associated with computations, and energy is being consumed during every evaluation step. *(ii)* Asynchronous notifications inform of energy exhaustion or computation termination. *(iii)* Mechanisms exist to transfer energy to or from computations; sup-

[1] Other names found in the literature for a similar concept are fuel [HF87], computron [Ray91, p. 102–103], teleclick [Mag96a] or metapill [A+97]

plying more energy to a computation gives the right to continue the computation, while removing energy from a computation acts as energy-based preemption. Even though the notion of energy is part of the semantics of *Quantum*, the programmer cannot create energy ex nihilo, but can only transfer it between computations via some primitives of the language. As a result, we were able to ensure a general principle for *Quantum*: given a finite amount of energy, any computation is finite.

Quantum generalises some approaches adopted in agent scripting languages to control resources [Mag96a, PS97]. Besides its resource-oriented foundations, *Quantum* is conceived for parallelism and distribution, while being independent of the actual primitives for parallelism and distribution, and of the memory model (central, distributed, with or without coherence).

This paper reports about our experience with designing *Quantum* and defining its semantics. Our requirement is to define the primitives that would allow us to control the energy consumption of distributed computations. We derived a solution from two different ideas: Haynes and Friedman's engines [HF87] can be extended to parallelism and distribution, while actor's sponsors [KH81] can be adapted to our energy view. For the sequential subset of the language, we adopt an applied call-by-value lambda-calculus [Plo75]. The semantics have been our driving force in designing *Quantum*. Our operational semantics takes the form of an abstract machine for parallel evaluation; it extends the CEK machine [FF86] with parallelism and the energy-related primitives. The choice of the semantic framework is a major help in defining simple primitives: the abstract machine hides some execution details and offers a suitable degree of atomicity, while it still offers a realistic model of parallelism.

This paper is organised of follows. We present the intuition of *Quantum* in Section 2 and define its semantics in Section 3. Section 4 contains several examples written in *Quantum*. Finally, Section 5 discusses related work and is followed by a conclusion.

2 Intuition of *Quantum*

In this section, we introduce the language *Quantum*, its constructs and their intuitive semantics, and the considerations that lead to its design. The primitives of *Quantum* that are specific to energy are displayed in the first half of Figure 1. In order to be usable, they need to be glued with other primitives for parallelism and communication, which may vary according to the programmer's taste. A particular selection is presented in the second half of Figure 1; they are briefly described below and they will be used in the examples of Section 4. Afterwards, we describe the three key ideas of *Quantum*: *groups*, *asynchronous notifications*, and *energy transfers*.

Energy Specific Primitives

$$\begin{array}{rcl}
\mathit{primitives} & ::= & \mathsf{call\text{-}with\text{-}group}(F, e, \varphi_e, \varphi_t) \\
 & & \mid\ \mathsf{pause}(g, \varphi_p) \\
 & & \mid\ \mathsf{awaken}(g, e) \\
g & \in & \mathit{Group} \\
e & \in & \mathit{Energy} \\
F & : & \mathit{Group} \times \mathit{Energy} \to \alpha \\
\varphi_e, \varphi_t, \varphi_p & : & \mathit{Group} \times \mathit{Energy} \to \mathit{void}
\end{array}$$

Language Specific Primitives

$$\begin{array}{rcl}
\mathit{primitives} & ::= & \mathsf{fork}(M)\ \mid\ \mathsf{suicide}() \\
 & & \mid\ \mathsf{channel}()\ \mid\ \mathsf{enqueue}(ch, V) \\
 & & \mid\ \mathsf{dequeue}(ch) \\
ch & \in & \mathit{Channel} \\
V & \in & \mathit{Value}
\end{array}$$

Figure 1: Abstract Syntax of Primitives

We use the term *task* to denote an evaluation thread created by the construct for parallelism. *Quantum* is independent of the primitives for parallelism and distribution. Parallel threads of evaluation may be created using Posix threads [IEEE], or higher-level constructs such as pcall [MR95, QD92] or future [Hal90, FF95, Mor96b]. In this paper, we adopt the construct fork which creates a parallel evaluation thread for its argument. A computation also has the ability to terminate by evaluating the expression suicide().

Our permanent concern when designing *Quantum* was to be able to compute in a distributed framework. Hence, we decided that *Quantum* would be independent of the memory model: so, real shared memory, shared memory simulated over a distributed memory [Mor96a], distributed causally coherent memory [Que94a] are memory models that may be adopted with *Quantum*. However, we need primitives to synchronise computations and to exchange information between them. We observed that asynchronous unbounded communication channels [KNY95] offered the appropriate level of abstraction. Figure 1 contains primitives to create channels, to add a value to a channel, and to remove a value from a channel.

2.1 Energy and Groups

Our goal is to be able to allocate resources to computations, and to monitor and to control their use as evaluation proceeds. In our view, it is essential to be notified of the *termination* of a computation so that, for instance, unconsumed resources can be transferred to a more suitable computation. Similarly, we want to be informed of the *exhaustion* of the resources allocated to a computation, so that for example more resources can be supplied.

In order to be notified of the termination or energy exhaustion of a computation, we need an entity that represents the computation. A *group* is an object that can be used to refer to a computation in a Quantum program. So, a group is associated with a computation that may be composed of several tasks proceeding in parallel; in turn, they can initiate subcomputations by creating subgroups. As a result, our computation model is hierarchical. A group is said to *sponsor* [KH81, Osb90b, Hal90] the computation it is associated with. Reciprocally, every computation has a sponsoring group, and so does every task.

At creation time, a group is given an *energy quota*. More specifically, a computation that evaluates the expression call-with-group$(F, e, \varphi_e, \varphi_t)$ under the sponsorship of a group g_1, creates a new first-class group g_2 that is allocated an initial quota of energy e and whose parent is g_1. Furthermore, it initiates a computation under the sponsorship of g_2 by applying F to g_2 and e; hence, the user function F receives a handle on its sponsoring group. As Quantum keeps track of resource consumption, the cost of g_2 creation and the energy e allocated to g_2 are deducted from the energy of g_1. The value of the call-with-group primitive is the value returned by the application of F.

2.2 Asynchronous Notifications

The semantics enforces the following principle: every computation consumes energy from its sponsoring group. Therefore, not only is a group perceived as a way of naming computations, but also it must be regarded as an *energy tank* for the computation. In addition, two events may be signalled during the lifetime of a group: *group termination* and *energy exhaustion* are asynchronously notified by applying the user functions (the *notifiers*) φ_t and φ_e, respectively[2]. A group is said to be terminated, when it has no subgroup and it does not sponsor any task; i.e. no more activity can be performed in the group. When group g_2 is terminated, the function φ_t is asynchronously called on g_2 to notify its termination, and the energy surplus of g_2 is transferred back to g_1. Note that calling φ_t is sponsored by g_1, i.e. the parent of g_2. Similarly, when a computation sponsored by g_2 requires more energy than available in g_2, the function φ_e is asynchronously called on g_2 to notify its energy exhaustion, also under the sponsorship of g_1, with transfer of the remaining energy of g_2 to g_1.

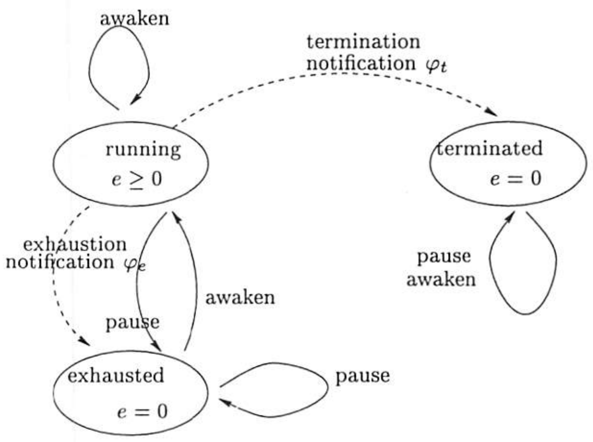


Figure 2: State Transitions

Figure 2 displays the state transition diagram for groups. At creation time, a group is in the running state, which means that the tasks that it sponsors can proceed as long as they do not require more energy than available. Asynchronous notifications are represented by dotted lines. Once a computation requires more energy than available in its sponsoring group, the state of its group changes to exhausted, and at the same time an asynchronous notification φ_e is run. When all subgroups and all tasks sponsored by a group terminate, its state becomes terminated, while the asynchronous notifier φ_t is called. Let us observe that the terminated state is a dead end in the state diagram; this guarantees the *stability* of the termination property: once a computation terminates, it is not allowed to restart (as the resource that it did not consume may have been reallocated). Here we should point out that the primitive fork can only create a thread in the group that sponsors its evaluation.

[2]Subscript t denotes termination, whereas subscript e denotes exhaustion.

2.3 Energy Transfers

Energy may be caused to flow between groups, independently of the group hierarchy, under the control of the user program. Two primitives operate on groups: pause and awaken. Intuitively, the primitive pause forces a running group *and its subgroups* into the exhausted state, and all the energy that was available in this hierarchy is transferred to the group that sponsored the pause action. The construct awaken(g, e) supplies a group g with some energy e, which is deducted from the group sponsoring the awaken action. In addition, if the group was in the exhausted state, it is changed to the running state; if the group is in a terminated state, awaken acts as a null operation. Let us observe the non-symmetric behaviours of pause and awaken: the former operates recursively on a group hierarchy, while the latter acts on a group and not its descendants. However, we might wish to awaken a hierarchy recursively, for instance when we wish to resume a paused parallel search. In particular, we might wish to resume the search with the energy distribution that existed when the hierarchy was paused. Unfortunately, such information is no longer available because groups are *memory-less*. It is therefore the programmer's responsibility to leave some information at pausing-time about the way a hierarchy should be awakened.
Not only does pause transfer energy, but it does also post a notification for each group in the tree. More precisely, pausing a group g with a notifier φ_p forces into the exhausted state each group g' in the hierarchy rooted by g; moreover, for each g', a task that applies φ_p on g' is created under the sponsorship of the parent of g'. Let us note that notifications are prevented to run as all groups in the hierarchy have been dried out (except the notification on the root g, which is sponsored by the parent of g and then might run). Once the root of the hierarchy is awakened, any notification sponsored by the root will be activated, and may decide to awaken the group it is applied on, and step by step, energy may be redistributed among the hierarchy.

3 The Language $\mathcal{Q}$uantum: Semantics

In this section, we present the semantics of $\mathcal{Q}$uantum using an abstract machine, called the $\mathcal{Q}$-machine. Figure 3 displays its state space. We can see that the primitives of Figure 1 appear in the set of terms Λ_Q. The core of Λ_Q is an applied call-by-value lambda calculus composed of abstractions, variables, applications, and constants [Plo75]. Let us note that fork and suicide only are essential syntax, whereas the other primitives appear as constants in the set *FConst*. Transition rules appear in Figures 4 to 10.
In the sequel, we adopt Barendregt's [Bar84] definitions and conventions on the lambda-calculus; in particular, n-ary functions should be understood as curried functions. We use the notation $f[x \to V]$ to denote the function f' such that $f'(x) = V$ and $f'(y) = f(y),\ \forall y \neq x$.
A configuration of the $\mathcal{Q}$-machine, represented as $\mathcal{M}$ in Figure 3, is a triple composed of a set of tasks, A set of groups and their associated information, and a set of channels and their contents. A *task*, represented by a pair $\langle C, g\rangle$, is an entity, sponsored by group g, that embodies a *computational state* C. In $\mathcal{Q}$uantum, tasks are anonymous and are not first-class values; instead, groups are reified as first-class objects as a mean to monitor and control computations. The function Γ associates each group g with a parent group, its current energy, its state, the two notifiers φ_e and φ_t, and the number of tasks and the set of subgroups that it sponsors. The hierarchy root is the *initial group*, and by convention, the parent of the initial group is represented by $\perp_g$.
Notifiers are closures with a signature $Group \to Energy \to Void$, which receive the group that is notified the event and its remaining energy. As notifications are asynchronous, they are not expected to return values, hence the void value returned. Channels are first-class values represented by $\langle \mathsf{ch}\ \alpha\rangle$, with α a location pointing to a queue in the queue store σ.
The computational state of a task is a CEK-configuration [FF86] represented as $\mathsf{Ev}\langle M, \rho, \kappa\rangle$ or $\mathsf{Ret}\langle V, \kappa\rangle$, respectively representing the evaluation of a term M in the environment ρ with a continuation κ, and the return of a value V to a continuation κ. The continuation κ is encoded by a data-structure, called *continuation code*.
Figure 4 displays the transition rules for the sequential purely functional subset of the language; for more detail, we refer the reader to [FF86].
As previously mentioned, the purpose of $\mathcal{Q}$uantum is to measure the resources used by computations. In order to be generic, we decided to associate the semantics with two *cost functions* $\mathcal{K}, \mathcal{K}_n$, giving each transition its cost in terms of energy.

> **Warning**. The cost of a transition is a function of the task involved in the transition and of the function Γ. For the sake of concision, we do not represent this dependency explicitly. We use the symbol

$M \in \Lambda_Q$	$::=$	$V_s \mid (M\ M) \mid (\mathsf{fork}\ M) \mid (\mathsf{suicide})$	(Term)
$V_s \in SValue$	$::=$	$c_s \mid x \mid (\lambda x.M)$	(Syntactic Value)
$V \in Value$	$::=$	$c \mid \ell \mid f_c \mid (\mathsf{cons}\ V\ V) \mid g \mid \langle \mathsf{ch}\ \alpha \rangle$	(Semantic Value)
$c_s \in SConst$	$::=$	$f \mid b$	(Syntactic Constant)
$f_c \in PApp$	$::=$	$(\mathsf{cons}\ V) \mid (\mathsf{enqueue}\ V) \mid$ $(\mathsf{call\text{-}with\text{-}group}\ V) \mid ((\mathsf{call\text{-}with\text{-}group}\ V)\ V)$ $(((\mathsf{call\text{-}with\text{-}group}\ V)\ V)\ V)$	(Partial Application)
$c \in Const$	$::=$	$c_s \mid d$	(Constant)
$b \in BConst$	$=$	$\{\mathsf{true}, \mathsf{false}, \mathsf{nil}, 0, 1, \ldots\}$	(Basic Constant)
$d \in Void$	$=$	$\{\mathsf{void}\}$	(Void Constant)
$f \in FConst$	$=$	$\{\mathsf{cons}, \mathsf{car}, \mathsf{cdr}, \mathsf{call\text{-}with\text{-}group},$ $\mathsf{channel}, \mathsf{enqueue}, \mathsf{dequeue}, \mathsf{pause}, \mathsf{awaken}\}$	(Functional Constant)
$x \in Vars$	$=$	$\{x, y, z \ldots\}$	(User Variable)
$\varphi \in Notifier$	$\subset$	$Closure$	(Notifier)
$\ell \in Closure$	$::=$	$\langle \mathsf{cl}\ \lambda x.M, \rho \rangle$	(Closure)
$\mathcal{M} \in Qconfig$	$::=$	$\langle T, \Gamma, \sigma \rangle$	(Q-Configuration)
$\Gamma \in GMap$	$:$	$Group \rightarrow GInfo$	(Group Mapping)
$i \in GInfo$	$::=$	$\langle g, e, s, \varphi_e, \varphi_t, n, g^* \rangle$	(Group Information)
$g \in Group$	$=$	$\{\bot_g\} \cup \{g_0, g_1, \ldots\}$	(Group)
$t \in Task$	$::=$	$\langle C, g \rangle$	(Task)
$C \in CoSt$	$::=$	$\mathsf{Ev}\langle M, \rho, \kappa \rangle \mid \mathsf{Ret}\langle V, \kappa \rangle$	(Computational State)
$\kappa \in CCode$	$::=$	$(\mathbf{init}) \mid (\kappa\ \mathbf{fun}\ V) \mid (\kappa\ \mathbf{arg}\ M\ \rho) \mid (\kappa\ \mathbf{rgroup})$	(Continuation code)
$s \in GState$	$::=$	$\mathsf{running} \mid \mathsf{exhausted} \mid \mathsf{terminated}$	(Group State)
T	$\subseteq$	$Task$	(Set of Tasks)
$q \in Queue$	$::=$	$\langle\rangle \mid \langle V \rangle \mid q \S q$	(Queue)
$\sigma \in QStore$	$:$	$Loc \rightarrow Queue$	(Queue Store)
$\alpha \in Loc$	$=$	$\{\alpha_0, \alpha_1, \ldots\}$	(Location)
$\rho \in Env$	$:$	$Vars \rightarrow Value$	(Env)
$n \in \mathbb{N}$			(Number of sponsored tasks)
$e \in Energy$	$\subseteq$	$\mathbb{N}$	(Energy)
$\mathcal{K}, \mathcal{K}_n$	$:$	$Task \times GMap \rightarrow Energy$	(Cost Function)

Figure 3: State Space

$\mathcal{K}$ to denote the value of the cost function for the task involved in the transition and a given Γ. For instance, in rule (*sequential*) of Figure 6, the task involved in the transition is $\langle C, g \rangle$; therefore, the symbol $\mathcal{K}$ stands for $\mathcal{K}(\langle C, g \rangle, \Gamma)$.

We also use the symbol $\mathcal{K}_n$ to denote the cost of a notification, i.e. the cost of rule (*termination*) or (*exhaustion*).

Rule (*sequential*) of Figure 6 states that if there exists a task $\langle C, g \rangle$ sponsored by a group g, such that a CEK-transition reduces C to C_1, then after transition the task becomes $\langle C_1, g \rangle$; the energy of g is decremented by the cost of the transition; the other tasks remain unchanged. Rule (*sequential*), as most other rules, assumes that the energy associated with g is greater than the sum of the transition cost and the notification cost, which is represented by the side-condition noted $(\star)$. This side-condition guarantees that after transition, we still have enough energy to post a notification if required.

We use the following notations for accessing and modifying components of the tuple associated with a group g. If $\Gamma(g) = \langle g_p, e, s, \varphi_e, \varphi_t, n, g^* \rangle$, then $\Gamma(g).p = g_p$, $\Gamma(g).e = e$, $\Gamma(g).s = s$, $\Gamma(g).n = n$, $\Gamma(g).g^* = g^*$. Updates are written as follows: $\Gamma[g.e := e_1]$ denotes $\Gamma[g \rightarrow \langle g_p, e_1, s, \varphi_e, \varphi_t, n, g^* \rangle]$, $\Gamma[g.s := s_1]$ denotes $\Gamma[g \rightarrow \langle g_p, e, s_1, \varphi_e, \varphi_t, n, g^* \rangle]$. Sometimes, we even combine both conventions so that $\Gamma[g.n := g.n - 1]$ should be read as $\Gamma[g \rightarrow \langle g_p, e, s, \varphi_e, \varphi_t, n-1, g^* \rangle]$.

As many cost models are conceivable, we decided to parameterise the semantics by the cost model. Figure 5 gives the definition of functions $\mathcal{K}, \mathcal{K}_n$, charging a unitary cost for every transition, in addition to

$$
\begin{array}{rcll}
\mathsf{Ev}\langle (M_1\ M_2), \rho, \kappa\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle M_1, \rho, (\kappa\ \textbf{arg}\ M_2, \rho)\rangle & (\textit{rator}) \\
\mathsf{Ev}\langle \lambda x.M, \rho, \kappa\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle\langle \mathsf{cl}\ \lambda x.M, \rho\rangle, \kappa\rangle & (\textit{lambda}) \\
\mathsf{Ev}\langle c, \rho, \kappa\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle c, \kappa\rangle & (\textit{constant}) \\
\mathsf{Ev}\langle x, \rho, \kappa\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle \rho(x), \kappa\rangle & (\textit{variable}) \\
\mathsf{Ret}\langle V, (\kappa\ \textbf{arg}\ M, \rho)\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle M, \rho, (\kappa\ \textbf{fun}\ V)\rangle & (\textit{rand}) \\
\mathsf{Ret}\langle V, (\kappa\ \textbf{fun}\ \langle \mathsf{cl}\ \lambda x.M, \rho\rangle)\rangle & \rightarrow_{cek} & \mathsf{Ev}\langle M, \rho[x \rightarrow V], \kappa\rangle & (\textit{apply}) \\
\mathsf{Ret}\langle (\mathsf{cons}\ V_1\ V_2), (\kappa\ \textbf{fun}\ \mathsf{car})\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle V_1, \kappa\rangle & (\textit{car}) \\
\mathsf{Ret}\langle (\mathsf{cons}\ V_1\ V_2), (\kappa\ \textbf{fun}\ \mathsf{cdr})\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle V_2, \kappa\rangle & (\textit{cdr}) \\
\mathsf{Ret}\langle V, (\kappa\ \textbf{fun}\ f)\rangle & \rightarrow_{cek} & \mathsf{Ret}\langle \delta(f, V), \kappa\rangle & (\delta)
\end{array}
$$

Figure 4: CEK Transitions

Unitary Cost Function

$\mathcal{K}(\langle \mathsf{Ret}\langle \varphi_t, (\kappa\ \textbf{fun}\ (((\mathsf{call\text{-}with\text{-}group}\ F)\ e)\ \varphi_e))\rangle, g\rangle, \Gamma) = e + 1$

$\mathcal{K}(\langle \mathsf{Ret}\langle e, (\kappa\ \textbf{fun}\ (\mathsf{awaken}\ g_1))\rangle, g\rangle, \Gamma) = e + 1 \quad \text{if } g \neq g_1, \Gamma(g_1).s \neq \mathsf{terminated}$

$\mathcal{K}(\langle C, g\rangle, \Gamma) = 1, \quad \text{otherwise}$

$\mathcal{K}_n = 1 \quad \text{(notification cost)}$

Soundness Constraints on Cost Functions $\mathcal{K}$ and $\mathcal{K}_n$

$\mathcal{K}(\langle \mathsf{Ret}\langle \varphi_t, (\kappa\ \textbf{fun}\ (((\mathsf{call\text{-}with\text{-}group}\ F)\ e)\ \varphi_e))\rangle, g\rangle, \Gamma) > e$

$\mathcal{K}(\langle \mathsf{Ret}\langle e, (\kappa\ \textbf{fun}\ (\mathsf{awaken}\ g_1))\rangle, g\rangle, \Gamma) > e \quad \text{if } g \neq g_1, \Gamma(g_1).s \neq \mathsf{terminated}$

$\mathcal{K}(\langle C, g\rangle, \Gamma) > 0, \text{otherwise}$

$\mathcal{K}_n \geq 1$

Figure 5: Cost Functions $\mathcal{K}, \mathcal{K}_n$

the quantity of energy transferred. Other definitions are acceptable as long as they satisfy the soundness constraints of Figure 5, which preserve the following principles: first, every computation step has a cost; second, transferring some energy costs this amount of energy at least.

In Figure 6, the rule for the construct (fork M) creates a new task to evaluate M with the same environment ρ and an initial continuation, resulting in an additional task in the current group. The construct (suicide) removes the current task from its sponsoring group; the $\mathcal{Q}$-machine behaves similarly when a void value is returned to the initial continuation.

Rules dealing with channels, which appear in Figure 7, are straightforward. The construct (channel) returns a new channel $\langle \mathsf{ch}\ \alpha\rangle$ with a newly allocated location α bound to an empty queue in the queue store. The primitive enqueue adds a value V at the end of the queue associated with the channel, while dequeue takes the first element of the queue. Note that the transition (*dequeue*) is allowed to be fired only if the queue is not empty; as a result, a task is not allowed to progress when trying to dequeue an element from an empty channel. For the sake of simplicity, we have decided not to associate a cost with such a "blocked" task. This could be easily overcome by adding a rule for the empty queue case which would charge its cost to the sponsoring group.

Figure 8 displays rules related to groups. Groups are created by evaluating (call-with-group $F\ e\ \varphi_e\ \varphi_t$), which results in the application of the partial application (((call-with-group F) e) φ_e) on φ_t. Then, rule (*make group*) creates a new group g_1 in a running state, whose parent is the sponsoring group g, with an energy e, with one sponsored task applying the closure F on g_1 and e. Following the soundness constraints of Figure 5, the sponsoring group g is

$\langle \{ \ \langle C, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle C_1, g\rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma\rangle \quad \text{if } C \rightarrow_{cek} C_1 \quad (\star)$ *(sequential)*

$\langle \{ \ \langle \mathsf{Ev}\langle(\mathsf{fork}\ M, \rho, \kappa\rangle, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, g\rangle, \quad \langle \mathsf{Ev}\langle M, \rho, (\mathbf{init})\rangle, g\rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n + 1], \sigma\rangle \quad (\star)$ *(parallel)*

$\langle \{ \ \langle \mathsf{Ev}\langle(\mathsf{suicide}), \rho, \kappa\rangle, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1], \sigma\rangle \quad (\star)$ *(suicide)*

$\langle \{ \ \langle \mathsf{Ret}\langle \mathsf{void}, (\mathbf{init})\rangle, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1], \sigma\rangle \quad (\star)$ *(init)*

Convention: $(\star) \equiv \Gamma(g).e \geq \mathcal{K} + \mathcal{K}_n$

Figure 6: Sequential and Parallel Evaluations

$\langle \{ \ \langle \mathsf{Ret}\langle(\mathsf{channel}), \kappa\rangle, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle(\mathsf{ch}\ \alpha), \kappa\rangle, g\rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma[\alpha \rightarrow \langle\rangle]\rangle \quad \text{with} \quad \alpha \notin DOM(\sigma) \quad (\star)$ *(channel)*

$\quad \langle \{ \ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{fun}\ (\mathsf{enqueue}\ \langle \mathsf{ch}\ \alpha\rangle))\rangle, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, g\rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma[\alpha := \sigma(\alpha) \ \S \ \langle V\rangle]\rangle \quad (\star)$ *(enqueue)*

$\langle \{ \ \langle \mathsf{Ret}\langle(\mathsf{ch}\ \alpha), (\kappa\ \mathbf{fun}\ \mathsf{dequeue})\rangle, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle V, \kappa\rangle, g\rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma[\alpha := q]\rangle \quad \text{if } \sigma(\alpha) = \langle V\rangle \ \S \ q \quad (\star)$ *(dequeue)*

Convention: $(\star) \equiv \Gamma(g).e \geq \mathcal{K} + \mathcal{K}_n$

Figure 7: Channels Related Operations

$\langle \{ \ \langle \mathsf{Ret}\langle \varphi_t, (\kappa\ \mathbf{fun}\ (((\mathsf{call\text{-}with\text{-}group}\ F)\ e)\ \varphi_e))\rangle, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle g_1, (((\kappa\ \mathbf{rgroup})\ \mathbf{arg}\ e, \emptyset)\ \mathbf{fun}\ F)\rangle, g_1\rangle \ \} \ \cup \ T, \Gamma_1, \sigma\rangle \quad (\star)$ *(make group)*

$\quad\quad \text{with } \Gamma_1 = \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1][g.g^* := g.g^* \cup \{g_1\}][g_1 \rightarrow \langle g, e, \mathsf{running}, \varphi_e, \varphi_t, 1, \emptyset\rangle],$

$\quad\quad g_1 \notin DOM(\Gamma)$

$\langle \{ \ \langle \mathsf{Ret}\langle V, (\kappa\ \mathbf{rgroup})\rangle g, \ \rangle \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle V, \kappa\rangle, g_1\rangle \ \} \ \cup \ T, \Gamma[g.e := g.e - \mathcal{K}][g.n := g.n - 1][g_1.n := g_1.n + 1], \sigma\rangle$ *(return group)*

$\quad\quad \text{with } g_1 = \Gamma(g).p \quad (\star)$

Convention: $(\star) \equiv \Gamma(g).e \geq \mathcal{K} + \mathcal{K}_n$

Figure 8: Groups Related Operations

$\langle \{ \ \langle C, g\rangle \ \} \ \cup \ T, \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle g, (((\mathbf{init})\ \mathbf{arg}\ e, \emptyset)\ \mathbf{fun}\ \varphi_e)\rangle, g_1\rangle \ \} \ \cup \ \{ \ \langle C, g\rangle \ \} \ \cup \ T, \Gamma_1, \sigma\rangle$ *(exhaustion)*

$\quad\quad \text{if } \Gamma(g) = \langle g_1, e, \mathsf{running}, \varphi_e, \varphi_t, n, g^*\rangle, \ g_1 \neq \bot_g, \ e < \mathcal{K}(\langle C, g\rangle) + \mathcal{K}_n, \ \Gamma(g_1).s = \mathsf{running}$

$\quad\quad \text{with } \Gamma_1 = \Gamma[g.s := \mathsf{exhausted}][g.e := 0][g_1.e := g_1.e + e - \mathcal{K}_n][g_1.n := g_1.n + 1]$

$\langle T, \{ \ (g \rightarrow \langle g_1, e, \mathsf{running}, \varphi_e, \varphi_t, n, g^*\rangle) \ \} \ \cup \ \Gamma, \sigma\rangle$

$\quad \rightarrow \quad \langle \{ \ \langle \mathsf{Ret}\langle g, (((\mathbf{init})\ \mathbf{arg}\ e, \emptyset)\ \mathbf{fun}\ \varphi_t)\rangle, g_1\rangle \ \} \ \cup \ T, \Gamma_1, \sigma\rangle$ *(termination)*

$\quad\quad \text{if } g_1 \neq \bot_g, \ \Gamma(g_1) = \langle g_2, e_1, s, \varphi_{e1}, \varphi_{t1}, n_1, g_1^*\rangle, \ n = 0, \ g^* = \emptyset, \ \Gamma(g_1).s = \mathsf{running}$

$\quad\quad \text{with } \Gamma_1 = \Gamma[g \rightarrow \langle g_1, 0, \mathsf{terminated}, \varphi_e, \varphi_t, n, g^*\rangle][g_1 := \langle g_2, e_1 + e - \mathcal{K}_n, s, \varphi_{e1}, \varphi_{t1}, n_1 + 1, (g_1^* \setminus \{g\})\rangle]$

Figure 9: Asynchronous Notifications

$$\begin{array}{ll}
\langle\{\ \langle \mathsf{Ret}\langle \mathsf{nil}, (\kappa\ \mathbf{fun}\ (\mathsf{pause}\ \varphi_p))\rangle, g\rangle\ \}\ \cup\ T, \Gamma, \sigma\rangle & \\
\quad \rightarrow\ \ \langle\{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, g\rangle\ \}\ \cup\ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma\rangle \quad (\star) & (\textit{pause group 1}) \\
\langle\{\ \langle \mathsf{Ret}\langle (\mathsf{cons}\ g_1\ g^*), (\kappa\ \mathbf{fun}\ (\mathsf{pause}\ \varphi_p))\rangle, g\rangle\ \}\ \cup\ T, \Gamma, \sigma\rangle & \\
\quad \rightarrow\ \ \langle\{\ \langle \mathsf{Ret}\langle (g^*\ \S\ \Gamma(g_1).g^*), (\kappa\ \mathbf{fun}\ (\mathsf{pause}\ \varphi_p))\rangle, g\rangle\ \}\ \cup\ \{\ t_1\ \}\ \cup\ T, \Gamma_1, \sigma\rangle & \\
\qquad\quad \text{if } g \neq g_1,\ \Gamma(g_1).s \neq \mathsf{terminated}\ \ (\star) & (\textit{pause group 2}) \\
\qquad \text{with } t_1 = \langle \mathsf{Ret}\langle g_1, (((\mathbf{init})\ \mathbf{arg}\ e, \emptyset)\ \mathbf{fun}\ \varphi_p)\rangle, g_2\rangle & \\
\qquad \text{with } \Gamma_1 = \Gamma[g.e := g.e + e - \mathcal{K}][g_1.e := 0][g_1.s := \mathsf{exhausted}][g_2.n := g_2.n + 1] & \\
\qquad \text{with } g_2 = \Gamma(g_1).p, e = \Gamma(g_1).e & \\
\quad \rightarrow\ \ \langle\{\ \langle \mathsf{Ret}\langle (g^*\ \S\ \Gamma(g_1).g^*), (\kappa\ \mathbf{fun}\ (\mathsf{pause}\ \varphi_p))\rangle, g\rangle\ \}\ \cup\ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma\rangle & (\textit{pause group 3}) \\
\qquad\quad \text{if } (g = g_1 \vee \Gamma(g_1).s = \mathsf{terminated})\ \ (\star) & \\
\langle\{\ \langle \mathsf{Ret}\langle e, (\kappa\ \mathbf{fun}\ (\mathsf{awaken}\ g_1))\rangle, g\rangle\ \}\ \cup\ T, \Gamma, \sigma\rangle & \\
\quad \rightarrow\ \ \langle\{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, g\rangle\ \}\ \cup\ T, \Gamma[g.e := g.e - \mathcal{K}][g_1.e := g_1.e + e][g_1.s := \mathsf{running}], \sigma\rangle & (\textit{awaken group 1}) \\
\qquad\quad \text{if } \ g \neq g_1,\ \Gamma(g_1).s \neq \mathsf{terminated}\ \ (\star) & \\
\quad \rightarrow\ \ \langle\{\ \langle \mathsf{Ret}\langle \mathsf{void}, \kappa\rangle, g\rangle\ \}\ \cup\ T, \Gamma[g.e := g.e - \mathcal{K}], \sigma\rangle & (\textit{awaken group 2}) \\
\qquad\quad \text{if } \ (g = g_1 \vee \Gamma(g_1).s = \mathsf{terminated})\ \ (\star) & \text{Convention: } (\star) \equiv \Gamma(g).e \geq \mathcal{K} + \mathcal{K}_n
\end{array}$$

Figure 10: Pause and Awaken Operations on Groups

deducted of the transition cost, which includes the energy e given to g_1: rule (*make group*) guarantees that no energy is generated during group creation.

Let us notice that F is applied on g_1, with a continuation (κ **rgroup**) indicating that the evaluation is performed under the sponsorship of a group. When a value is returned to the continuation code **rgroup** as in (*return group*), the task leaves the sponsorship of its group: the task that was sponsored by g now becomes sponsored by its parent g_1; the numbers of tasks sponsored by g and g_1 are updated accordingly.

Rules controlling asynchronous notifications appear in Figure 9. If the energy available in a group g is smaller that the sum of the energy required by a task and the energy for a notification, rule (*exhaustion*) changes the state of g to exhausted, and creates a new task applying the notifier φ_e on g and the remaining energy e. The notification is executed under the sponsorship of the parent group g_1, and the energy e is transferred to g_1.

Asynchronous termination detection follows a similar pattern: if a group g does not sponsor any task and has no subgroup, a notifier φ_t is applied on g and the remaining energy e under the sponsorship of the parent group g_1; the remaining energy is also transferred to g_1.

As notifications are executed under the sponsorship of the parent of the group terminating or being exhausted, care should be taken not to apply these rules to the root of the hierarchy, which is expressed by the condition $g_1 \neq \bot_g$.

A notifier is defined as a user function. Evaluating a call to a notifier is a notification. Posting a notification is creating a task that performs a notification. Like any other transition, rules (*exhaution*) and (*termination*) are given a cost; for the sake of simplicity, it is defined as $\mathcal{K}_n$. The side-condition ($\star$) used in all rules but (*exhaution*) and (*termination*) ensures that enough energy is left in a group to post a notification.

Notification rules transfer energy from the notified group to its parent, avoiding energy loss. Notifications allow the user program to *observe* semantically-caused energy transfers between groups. Even though the user code is given access to the amount of energy transferred, energy accounting remains strictly under control of the semantics, which guarantees the safeness of the approach.

The semantics of pause and awaken is displayed in Figure 10. The primitive pause requires two arguments: a notification function φ_p and a list of groups to be paused. For each group g_1 of the list, rule (*pause group 2*) sets the state of g_1 to exhausted, transfers its remaining energy e to the group g sponsoring the pause action, creates a task applying the notifier φ_p on g_1 and e under the sponsorship of g_2 the parent of g_1, and adds the subgroups of g_1 to the list of groups remaining to be processed.

Special care is taken in rule (*pause group 3*) to avoid pausing the current group or to avoid setting a terminated group to the exhausted state. In rule (*pause group 1*), we see that the primitive pause returns a void value as it is used for its side-effect on group energies.

The primitive awaken takes the group to be awakened and the energy to be transferred in arguments. Assuming the sponsoring group g has enough energy, rule (*awaken group 1*) decrements its energy, increments the energy of the awakened group g_1, and sets it to the state running. Again care is taken to avoid awakening a terminated group. Let us observe again that the user specifies the amount of energy to be transferred but accounting is strictly performed at the semantic level.

Definition 1 displays the evaluation *relation* of the language. Evaluation starts with an initial configuration, composed of the initial group g_0, a queue store containing a location α_0 aimed at receiving all values generated by the computation, and an initial task; this task evaluates the program in an empty environment, and with a continuation accumulating the results obtained in location α_0. The evaluation relation associates a program with all the possible final results that can be accumulated in α_0.

Definition 1 (Evaluation Relation)
$eval_{\mathcal{K},\mathcal{K}_n}(M,e) = V$ if $\exists\langle T,\Gamma,\sigma\rangle$, such that $\langle\{\langle \mathsf{Ev}\langle M,\emptyset,\kappa_0\rangle,g_0\rangle\},\Gamma_0,\sigma_0\rangle \rightarrow^* \langle T,\Gamma,\sigma\rangle$, with $V\in\sigma(\alpha_0)$ and $Final(\langle T,\Gamma,\sigma\rangle)$, and with:

$$\begin{aligned}
\Gamma_0 &= \{g_0 \rightarrow \langle \bot_g, e, \varphi_{e0}, \varphi_{t0}, 1, \langle\rangle\rangle\}\\
\sigma_0 &= \{\alpha_0 \rightarrow \langle\rangle\}\\
\kappa_0 &= ((\mathbf{init})\mathbf{fun}\ (\mathsf{enqueue}\ \langle \mathsf{ch}\ \alpha_0\rangle))\\
Final(\langle T,\Gamma,\sigma\rangle) &\equiv \nexists\langle T',\Gamma',\sigma'\rangle,\\
&\qquad \langle T,\Gamma,\sigma\rangle \rightarrow \langle T',\Gamma',\sigma'\rangle\\
\varphi_{e0}=\varphi_{t0} &= \langle \mathsf{cl}\ \lambda g.\lambda e.\mathsf{void},\emptyset\rangle
\end{aligned}$$

□

Let us note that the evaluation relation is parameterised by the cost functions $\mathcal{K},\mathcal{K}_n$ and by the initial energy quota e given to the $\mathcal{Q}$-machine. The initial group has no parent, receives the initial energy quota, sponsors the initial task; the notification functions are arbitrary because they are never called, as seen in rules (*exhaustion*) and (*termination*).

We establish the soundness of the semantics with respect to energy by the next two propositions.

Proposition 2 For any cost function satisfying the constraints of Figure 5, total energy decreases as evaluation proceeds. □

Corollary 3 For any cost function satisfying the constraints of Figure 5, and for a finite positive initial energy, any computation is finite. □

4 Examples

In this Section, we adopt Scheme syntax [RC91] and present some examples using our primitives.

4.1 Energy Critical Section

Even though $\mathcal{Q}$uantum substantially differs from Scheme, it is expressive enough to model first-class mutable boxes. Figure 11 displays the code for mutable boxes in $\mathcal{Q}$uantum, where a mutable box is represented by a channel. Functions *deref* and *setref!* maintain the invariant that the channel contains one and only one value, by first dequeuing the current value, and then enqueuing another one. Simultaneous accesses to a same box are protected by the atomicity of the primitive *dequeue*.

```
(define (makeref V)
  (let ((c (channel)))
    (enqueue c V)
    c))
(define (deref c)
  (with-enough-energy
    (let ((v (dequeue c)))
      (enqueue c v)
      v)))
(define (setref! c v)
  (with-enough-energy
    (let ((old (dequeue c)))
      (enqueue c v)
      old)))
```

Figure 11: First-Class mutable boxes

However, a program could run out of energy after having read a value and before having stored the new one into the channel: this would leave the box in a inconsistent state, unusable by other tasks. Therefore, we must be sure that an exhaustion notification cannot occur between these two operations. This kind of "energy-critical section" is implemented by creating a group which receives the amount of energy minimal-energy required to perform both operations (Figure 12).

Let us notice that such a group does not prevent the program to be paused from outside. However, if such a pausing action occurs, it can only be caused

```
(define-syntax with-enough-energy
  (syntax-rules ()
    ((with-enough-energy form ...)
     (call-with-group (lambda (g e)
                        form ...)
                      minimal-energy
                      refill-handler
                      ignore-handler))))
(define (refill-handler g e)
  (awaken g (+ e 1)))
(define (ignore-handler g e)
  (suicide))
```

Figure 12: Energy-critical Section

by the user's program. It is his role to ensure that a paused group does not leave objects such as boxes in an inconsistent state.

4.2 Monitoring Computations

In traditional computing, we have no tool to tell us whether a computation is running or what is the amount of work done by a given task. Such tools usually exist at the operating-system level, but deal with "processes" and not with tasks, and they do not take into account tasks executed on remote hosts. Figure 13 displays the code of a probe, which updates the content of a box with the amount of energy already consumed by a computation generated by a *thunk*. When the computation ends, the box is updated with a pair indicating the total consumption of the *thunk*.

```
(define (probe box thunk unit)
 (call-with-group
  (lambda (g e) (thunk))
  unit
  (lambda (g e)
    (setref! box (+ unit (deref box)))
    (awaken g (+ unit e)))
  (lambda (g e)
    (setref! box (cons 'sum (- (deref box) e))))))
```

Figure 13: Probe

4.3 Controlling Computations

Quantum offers the possibility to control computations in a refined way. It is possible to stop a (distributed) computation, to suspend and resume it later, or to adjust it level energy, which introduces a form of energy-based priority. All three operations are implemented using *pause* and *awaken*.

The function *suspend* temporarily pauses a group hierarchy. The hierarchy will be resumed by awakening its root, which will awaken its subgroups step by step using the notifiers left by *pause*. Note that the condition in the notifier prevents to awaken the root of the hierarchy immediately after the root has received the notification.

On the contrary, the function *kill* pauses a hierarchy without leaving any opportunity to resume it, unless the programmer has explicitly kept handles on the groups that belong to the hierarchy, and explicitly awakens them.

Last, *adjust-energy* pauses and immediately resumes a hierarchy with a quota of energy which is proportional to the one it had before. The energy transfer that occurs during this operation is worth noticing: the group calling *adjust-energy* will be credited of the energy of hierarchy before adjustment, while the energy of the hierarchy after adjustment is deducted from the parent of the root. In order to guarantee that *awaken* is executed after *pause*, we introduce an explicit synchronisation by the channel *go*.

```
(define (suspend group)
  (pause (lambda (g e)
           (if (not (eq? g group))
               (awaken g e)))
         (list group)))
(define (kill group)
  (pause (lambda (g e)
           (suicide))
         (list group)))
(define (adjust-energy group coefficient)
  (let ((go (channel)))
    (pause (lambda (g e)
             (if (eq? g group)
                 (dequeue go))
             (awaken g (* e coefficient)))
           (list group))
    (enqueue go #t)))
```

Figure 14: Pause and Awaken of Computations

4.4 A Service Provider

Figure 15 displays the code of a service provider, making use of Wright and Duba's pattern-matching macro [WD95]. A communication channel *subscription-channel* is publicly advertised as the entry point to the service provider. A user is allowed to subscribe to the service by giving its name (possibly authenticated by a specialised protocol), some electronic cash, and a channel to which the service provider answers. The service provider creates a new communication channel that is returned to the user and that will be used as an access point to resources offered by the service provider. This access point is served by a *request-server*, whose operations are sponsored by a group that has received an initial amount of energy corresponding to the electronic cash transmitted at subscription time.

The user can submit jobs to the *request-server*, which in turn creates a new group to sponsor the evaluation of *job* and returns it to the user. This group can be used by the user to pause, kill, or restart a computation. Let us observe that the user is never given a handle either on the group initially created with his electronic cash, or on the group sponsoring the administration program. Thanks to lexical scoping, these groups can be hidden, and security is insured because nobody will be able to pause and steal energy from such groups.

When the account of a user is exhausted, a message is sent to the user, who gets the opportunity to transfer more electronic cash to his account via a message 'pay. This request is sent on the publicly advertised channel, *subscription-channel*, and might need some authentication protocol.

The service providers may also offer a demonstration account which will be usable for a fixed quota of energy energy-quota-for-free-demo and which may offer restricted facilities only. This program can be extended by offering the possibility to close an account and to refund the electronic cash corresponding to the remaining energy.

5 Discussion and Related Work

For the sake of clarity, we have presented a simple cost model that measures the cost of computing. In practice, we should regard energy as a multidimensional (vector) datastructure of budgetised resources. Such an extended model can take into account time, geographical expansion, file access permission, memory size, number of messages, number of parallel tasks, ... In such an extended model, once a resource is exhausted (or a computation ends), there is an asynchronous notification; the primitives awaken and pause can supply or remove a given resource.

Our notion of group is at the intersection of two different ideas: Haynes and Friedman's engines and Kornfeld, Hewitt, and Osborne's sponsors, which we develop below.

Haynes and Friedman [HF84, HF87] introduce the *engine* facility to model timed preemption; variants can also be found in [Dyb87, Eis88, Sit94]. Engines differ from our groups in a number of ways. Engines are defined in a *sequential* framework and are used to simulate multiprogramming. Since engines do not deal with parallelism, they do not offer control facilities such as pause and awaken. Another major difference is that a given engine can be executed several times, while a group can only be executed once. Using continuation terminology, engines are "multi-shot", while groups are "single-shot" [BWD96]. A group is a name and an energy tank for a computation, but, unlike an engine, it does not embody its continuation. Our decision to design "single-shot" groups is motivated as follows. The ability to restart several times a same computation is an unrealistic feature for a distributed language because the computation may be composed of several tasks distributed over the net. Haynes and Friedman also propose *nested engines*, i.e. engines that can create other engines. In their approach, nested engines have the same temporal vision of the world, because each computation consumes ticks, i.e. energy quanta, from parent engines (direct *and* indirect). On the contrary, groups offer more a distributed vision of the world, because groups are tanks, from which local tasks consume energy.

Kornfeld and Hewitt's sponsors [KH81], Osborne's enhanced version of them [Osb90a, Osb90b, Hal90], and subsequently Queinnec's groups [Que94b, QD92], also allow the programmer to control hierarchies of computations in a parallel setting. Osborne's sponsors are entities that give attributes, such as priority, to tasks, which can inherit attributes from several sponsors. A combining rule yields the effective attributes of a task, and then determines the resources allocated to the task. If the group hierarchy changes, priorities should be recomputed, which can be costly, especially in a distributed environment. With Quantum groups, scheduling of a task is only decided by examining the energy available in its only sponsoring group, which is local. Furthermore, priority is a difficult notion to grasp in a heterogeneous environment, while resources are more intuitive. Queinnec's Icsla language has a notion of group which substantially differs from the one presented here. As Icsla is energy-less, pausing a group does not collect energy and can be performed lazily. Also, Icsla does not have any of the notifications of Quantum. Let us observe that termination notification is a generali-

```
(define subscription-channel (channel))
(define (service-provider subscription-channel)
  (let loop ()
    (let ((subscription (dequeue subscription-channel)))
      (fork (process-subscription subscription))
      (loop))))
(define (process-subscription subscription)
 (match subscription
  (('subscribe name ecash answer)
   (let ((private-channel (channel))
         (energy (ecash->energy ecash)))
    (call-with-group (lambda (g e)
                        (register name g private-channel)
                        (enqueue answer private-channel)
                        (request-server private-channel g))
                      energy
                      (lambda (g e)
                        (enqueue answer
                                 "Account exhausted"))
                      ignore-handler)))

  (('pay name ecash)
   (let ((group (get-group name)))
     (awaken group (ecash->energy ecash))))

  (('free-demo name answer)
   (call-with-group (lambda (g e)
                      (let ((private-channel (channel)))
                       (enqueue answer private-channel)
                       (request-server private-channel g)))
                    energy-quota-for-free-demo
                    (lambda (g e)
                      (enqueue answer
                               "Demo account exhausted"))
                    ignore-handler))
  (else 'discard))
 (suicide))
```

```
(define (request-server channel sponsoring-group)
 (let ((message (dequeue channel)))
  (fork (request-server channel sponsoring-group))
  (match message
   (('submit job answer)
    (call-with-group (lambda (g e)
                         (add-group! sponsoring-group g)
                         (enqueue answer `(created ,g))
                         (job))
                       1
                       refill-handler
                       (lambda (g e)
                         (remove-group! sponsoring-group g)
                         (enqueue answer `(done ,g)))))
   (('pause group answer)
    (if (member group (subgroups sponsoring-group))
        (begin
          (suspend group)
          (enqueue answer `(paused ,group)))
        (enqueue answer `(unknown ,group))))
   (('kill group answer)
    (if (member group (subgroups sponsoring-group))
        (begin
          (remove-group! sponsoring-group group)
          (kill group)
          (enqueue answer `(killed ,group)))
        (enqueue answer `(unknown ,group))))
   (('restart group answer)
    (if (member group (subgroups sponsoring-group))
        (begin
          (awaken group 1)
          (enqueue answer `(restarted ,group)))
        (enqueue answer `(unknown ,group))))
   (else 'discard))
  (suicide)))
```

Figure 15: Service Provider (1)

sation of `unwind-protect` [Ste90]. Hieb and Dybvig [HD90] `spawn` operator returns a controller, which can be invoked to suspend or restart part of a computation tree; their approach relies on a notion of partial continuation.

The mobile agent community deals with the problem of controlling distributed mobile computations, called *agents*. The most widespread agent systems are Telescript [Mag96a, Mag96b], Tacoma [JvRS95a, JvRS95b], Agent-TCL [Gra96], and Ara agents [Pei97, PS97]. Most of them have a notion of energy: Telescript agents have "permits" in terms of teleclicks [Mag96a, Mag96b], Ara agents [Pei97, PS97] are equipped with resource accounts called *allowances*, Erlang agents [A$^+$97] rely on metapills. However, very few of them are able to control resources in a similar way as $\mathcal{Q}$uantum.

Quoting [Mag96b], "if the agent exceeds any of its quantitative limits, the engine destroys the agent unceremoniously. No grace period is extended". Our model is more general than the Telescript approach as it allows us to drive computations using pause and awaken and to monitor them using asynchronous notifications. Also, our model supports agents that perform parallel and distributed computations, what is usually called multi-agent systems. Arthursson *et al.* [A$^+$97] follow an approach similar to Telescript.

Allowances in Ara agents [PS97, Pei97] are similar to our groups [PS97, p. 6]; as indicated by Peine, they are a recent concept, not complete yet. Several agents can share the same allowance, also called a group. Agents can transfer resources explicitly between accounts. In Ara, agents can be suspended or reactivated: these actions however are performed on agents directly and not on groups [PS97, p. 23]; as a result hierarchical computation cannot be controlled as in $\mathcal{Q}$uantum. Ara agents are not notified of an exhaustion [PS97, p. 35]; Peine consider that this is not a severe problem as an agent may enquire about the existence of a resource. We believe that this argument is not valid in parallel/distributed computing because another parallel computation might consume the resource.

Our semantic is sound because it prevents generating energy. Furthermore, our language provides the means to enforce security in different ways: *(i)* energy cannot be generated, but can only be transferred between computations; all "accounting" operations remain under absolute control of the semantics; *(ii)* groups are the only handle to control computations, and lexical scoping guarantees that groups will be visible only where the programmer wishes them to be, *(iii)* there is no primitive that returns the group in which the user code is running, which ensures that user code cannot control its sponsoring group, and hence it cannot control tasks running in parallel with it, unless explicitly passed a handle to their sponsoring group. Security is an important issue in distributed agent-style applications. Using Quantum primitives, there is nothing that prevents users from erroneously transferring resources between groups, or making a group accessible to and then preemptable by another task. However, we believe that some static analysis [VS97] would be able to detect whether a group might become preempted if made accessible in a public data structure for instance.

6 Conclusion

In this paper, we present the language Quantum, whose purpose is to monitor and control resource consumption in a parallel and distributed framework. The semantics uses an abstract notion of energy, but it can be applied to control computation time, memory usage, message sending, file access permission, etc. A companion paper describes a distributed implementation of Quantum built on top of a message-passing library [MQ97]. A refinement of the semantics introduces explicit localities in the spirit of [Ama97, FGLMR96, Mor96a].

Quantum is a language that is aimed at agent application implementers, who are in need of controlling and bounding the resources needed by their agents. It is also useful to implementers of services who need to monitor visiting mobile agents. Also, it provides primitives to build "any-resource" algorithms, applying the idea "any-time" algorithms [DB88] to any form of resource.

Quantum is the core of a consumption-oriented language which is particularly suitable to program over the Internet. In the future, we plan to investigate a fault-tolerant version of the language, which would be energy aware.

7 Acknowledgement

Luc Moreau was partially supported by EPSRC GR/K30773 and EC project reference ERB 4050 PL 930186; Christian Queinnec was partially supported by GDR-PRC de Programmation du CNRS and EC project reference ERB 4050 PL 930186. Thanks to Jonathan Dale and the anonymous referees for their useful comments. Andreas Kind and Jean-Pierre Briot mentioned a link between Quantum and anytime functions.

Bibliography

[A+97] J. Arthursson et al. A Platform for Secure Mobile Agents. In *The Second International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 109–120, April 1997.

[Ama97] R. M. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *COORDINATION 97, LNCS*, 1997.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, second edition, 1984.

[BWD96] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representig Control in the Presence of One-Shot Continuations. In *ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, pages 99–107, Philadelphia, Pennsylvania, May 1996.

[DB88] T. L. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, Minneapolis, Minnesota, 1988.

[DD97] J. Dale and D. DeRoure. Towards a Framework for Developing Mobile Agents for Managing Distributed Information Resources. Technical Report M97/1, University of Southampton, February 1997.

[Dyb87] R. K. Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[Eis88] M. Eisenberg. *Programming in Scheme.* The Scientific Press, 507 Seaport Court. Redwood City. CA 94063-2731, 1988.

[FF86] M. Felleisen and D. P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).

[FF95] C. Flanagan and M. Felleisen. The Semantics of Future and Its Use in Program Optimization. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1995. Technical Reports 238, 239, Rice University, 1994.

[FGLMR96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR'96*, LNCS 1119, pages 406–421, Pisa, Italy, 1996. Springer-Verlag.

[Gra96] R. S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, July 1996. http://www.cs.dartmouth.edu/~agent/papers/index.html.

[Hal90] R. H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. In *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, LNCS 441, pages 2–57. Springer-Verlag, 1990.

[HD90] R. Hieb and R. K. Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.

[HF84] C. T. Haynes and D. P. Friedman. Engines Build Process Abstractions. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 18–24. ACM, 1984.

[HF87] C. T. Haynes and D. P. Friedman. Abstracting Timed Preemption with Engines. *Comput. Lang.*, 12(2):109–121, 1987.

[IEEE] IEEE. *IEEE P1003.1c/D10 Draft Standard for Information Technology – Portable Operating Systems Interface (POSIX)*, September 1994.

[JvRS95a] D. Johansen, R. van Renesse, and F. B. Schneider. An Introduction to the TACOMA Distributed System Ver 1. Technical Report 95–23, Department of Computer Science, University of Troms, Norway, 1995. http://www.cs.uit.no/DOS/Tacoma/Publications.html.

[JvRS95b] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In *5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa, USA, 1995. Also available as Technical Report TR94-1468, Department of Computer Science, Cornell University. http://www.cs.uit.no/DOS/Tacoma/Publications.html.

[KH81] W. A. Kornfeld and C. E. Hewitt. The Scientific Community Metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

[KNY95] N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, LNCS 983, pages 225–242. Springer-Verlag, 1995.

[Mag96a] General Magic. Telescript Technology: Mobile Agents. http://www.genmagic.com/Telescript/Whitepapers/wp4/white paper-4.html, 1996.

[Mag96b] General Magic. Telescript technology: The foundation for the electronic marketplace. http://www.genmagic.com/Telescript/Whitepapers/wp1/white paper-1.html, 1996.

[Mor96a] L. Moreau. Correctness of a Distributed-Memory Model for Scheme. In *Second International Europar Conference (EURO-PAR'96)*, LNCS 1123, pages 615–624, Lyon, France, August 1996. Springer-Verlag.

[Mor96b] L. Moreau. The Semantics of Scheme with Future. In *In ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 146–156, Philadelphia, Pennsylvania, May 1996.

[MQ97] L. Moreau and C. Queinnec. Distributed computations driven by resource consumption. Technical report, University of Southampton, 1997.

[MR95] L. Moreau and D. Ribbens. The Semantics of pcall and fork. In *PSLS 95 - Parallel Symbolic Langages and Systems*, LNCS 1068, pages 52–77, Beaune, France, October 1995. Springer-Verlag.

[Osb90a] R. B. Osborne. Speculative Computation in Multilisp. In *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, LNCS 441, pages 103–137. Springer-Verlag, 1990.

[Osb90b] R. B. Osborne. Speculative Computation in Multilisp. An Overview. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 198–208, Nice, France, June 1990.

[Pei97] H. Peine. An Introduction to Mobile Agent Programming and the Ara System. Technical Report ZRI report 1/9, Dept. of Computer Science, University of Kaiserslautern, Germany, 1997. http:// www.uni-kl.de/AG-Nehmer/Ara/.

[Plo75] G. D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[PS97] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proc. of the First International Workshop on Mobile Agents MA'97*, LNCS 1219, Berlin, Germany, April 1997. Springer-Verlag. http:// www.uni-kl.de/AG-Nehmer/Ara/.

[QD92] C. Queinnec and D. DeRoure. Design of a Concurrent and Distributed Language. In *Parallel Symbolic Computing: Languages, Systems and Applications*, LNCS 748, pages 234–259, Boston, Massachussetts, October 1992. Springer-Verlag.

[Que94a] C. Queinnec. Locality, Causality and Continuations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[Que94b] C. Queinnec. Sharing mutable objects and controlling groups of tasks in a concurrent and distributed language. In *Proceedings of the Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, LNCS 700, pages 70–93, Sendai (Japan), November 1994. Springer-Verlag.

[Ray91] E. Raymond. *The New Hacker's Dictionary.* MIT Press, 1991.

[RC91] J. Rees and W. Clinger. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[Sit94] D. Sitaram. *Models of Control and Their Implications for Programming Language Design.* PhD thesis, Rice University, Houston, Texas, April 1994.

[Ste90] G. L. Steele, Jr. *Common Lisp. The Language.* Digital Press, second edition, 1990.

[VS97] D. Volpano and G. Smith. A type-based approach to program security. In *Theory and Practice of Software Development (TAPSOFT'97)*, LNCS 1214, pages 607–621, Lille, France, April 1997. Springer-Verlag.

[WD95] A. W. Wright and B. F. Duba. Pattern Matching for Scheme. Technical report, Rice University, Hoston, TX 77251-1892, May 1995.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), June 1995.

Domains of Concern in Software Architectures and Architecture Description Languages

Nenad Medvidovic and David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425, U.S.A.
{neno,dsr}@ics.uci.edu

Abstract

Software architectures shift the focus of developers from lines-of-code to coarser-grained elements and their interconnection structure. Architecture description languages (ADLs) have been proposed as domain-specific languages for the domain of software architecture. There is still little consensus in the research community on what problems are most important to address in a study of software architecture, what aspects of an architecture should be modeled in an ADL, or even what an ADL is. To shed light on these issues, we provide a framework of architectural domains, or areas of concern in the study of software architectures. We evaluate existing ADLs with respect to the framework and study the relationship between architectural and application domains. One conclusion is that, while the architectural domains perspective enables one to approach architectures and ADLs in a new, more structured manner, further understanding of architectural domains, their tie to application domains, and their specific influence on ADLs is needed.

Keywords — *software architecture, architecture description language, domain, domain-specific language, architectural domain*

1. Introduction

Software architecture is an aspect of software engineering directed at reducing costs of developing applications and increasing the potential for commonality among different members of a closely related product family [PW92, GS93]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. This enables developers to abstract away the unnecessary details and focus on the "big picture:" system structure, high level communication protocols, assignment of software components and connectors to hardware components, development process, and so on.

Many researchers have realized that, to obtain the benefits of an architectural focus, software architecture must be provided with its own body of specification languages and analysis techniques [Gar95, GPT95, Wolf96]. Such languages are needed to demonstrate properties of a system upstream, thus minimizing the costs of errors. They are also needed to provide abstractions adequate for modeling a large system, while ensuring sufficient detail for establishing properties of interest. A large number of *architecture description languages* (ADLs) has been proposed, each of which embodies a particular approach to the specification and evolution of an architecture. Examples are Rapide [LKA+95, LV95], Aesop [GAO94], MetaH [Ves96], UniCon [SDK+95], Darwin [MDEK95, MK96], Wright [AG94a, AG94b], C2 [MTW96, MORT96, Med96], and SADL [MQR95]. Recently, initial work has been done on an architecture interchange language, ACME [GMW95, GMW97], which is intended to support mapping of architectural specifications from one ADL to another, and hence provide a bridge for their different foci and resulting support tools.

There is still very much a lack of consensus in the research community on what an ADL is, what aspects of an architecture should be modeled by an ADL, and what should be interchanged in an interchange language. This divergence has resulted in a wide variation of approaches found in this first generation of ADLs. Perhaps even more significantly, there is a wide difference of opinions as to what problems are most important to address in a study of software architecture.

In our previous research, we have provided a foundation for understanding, defining, classifying, and comparing ADLs [Med97, MT97]. In this paper, we build upon those results by identifying and characterizing *architectural domains*, the problems or areas of concern that need to be addressed by ADLs. Understanding these domains and their properties is a key to better understanding the needs of software architectures, architecture-based development, and architectural description and interchange. A study of architectural domains is also needed to guide the development of next-generation ADLs.

This paper presents a framework of architectural domains. We demonstrate that each existing ADL currently supports only a small subset of these domains, and we discuss possible reasons for that. Finally, we consider the relationship between architectural domains and application domains.

While we draw from previous ADL work and reference a number of ADLs, the most significant contribution of this paper is the framework of architectural domains. It provides structure to a field that has been approached largely in an ad-hoc fashion thus far. The framework gives the architect a sound foundation for selecting an ADL and orients discourse away from arguments about notation and more towards solving important engineering problems.

The remainder of the paper is organized as follows. Section 2 provides a short discussion of ADLs. Section 3 presents and motivates each architectural domain, while Section 4 discusses the support for architectural domains in existing ADLs. Section 5 expounds on the relationship between application domains and architectural domains. Discussion and conclusions round out the paper.

2. Overview of ADLs

To properly enable further discussion, several definitions are needed. In this section, we define software architectures, architectural styles, and ADLs.[1] We categorize ADLs, differentiate them from other, similar notations, and discuss examples of use of ADLs in actual projects. Finally, we provide a short discussion on our use of the terms "architecture" and "design."

2.1. Definitions of Architecture and Style

There is no standard definition of architecture, but we will use as our working definition the one provided by Garlan and Shaw [GS93]:

> *[*Software architecture *is a level of design that] goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.*

Architectural style is "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done" [SC96].

2.2. Definition of ADLs

Loosely defined, "an *ADL* for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module" [Ves93]. ADLs provide both a concrete syntax and a conceptual framework for modeling a software system's *conceptual* architecture.

The building blocks of an architectural description are
- *components* - units of computation or data stores;
- *connectors* - architectural building blocks used to model interactions among components and rules that govern those interactions; and
- *architectural configurations* - connected graphs of components and connectors that describe architectural structure.

An ADL must provide the means for their *explicit* specification; this criterion enables one to determine whether or not a particular notation is an ADL. In order to infer any kind of information about an architecture, at a minimum, *interfaces* of constituent components must also be modeled formally. Without this information, an architectural description becomes but a collection of (interconnected) identifiers.

An ADL's conceptual framework typically subsumes a formal semantic theory. That theory is part of the ADL's underlying framework for characterizing architectures; it influences the ADL's suitability for modeling particular kinds of systems (e.g., highly concurrent systems) or particular aspects of a given system (e.g., its static properties). Examples of formal specification theories are Petri nets [Pet62], Statecharts [Har87], partially-ordered event sets [LVB+93], communicating sequential processes (CSP) [Hoa85], model-based formalisms (e.g., *CH*emical *A*bstract *M*achine [IW95], Z [Spi89]), algebraic formalisms (e.g., Obj [GW88]), and axiomatic formalisms (e.g., Anna [Luc87]).

Finally, even though the suitability of a given language for modeling architectures is independent of whether

1. This section is condensed from a detailed exposition on ADLs given in [Med97] and [MT97], where we provided a definition of ADLs and devised a classification and comparison framework for them.

and what kinds of *tool support* it provides, an accompanying toolset will render an ADL both more usable and useful. Furthermore, capabilities provided by such a toolset are often a direct reflection of the ADL's intended use.

2.3. Categorizing ADLs

Existing languages that are commonly referred to as ADLs can be grouped into three categories, based on how they model configurations:

- *implicit configuration languages* model configurations implicitly through interconnection information that is distributed across definitions of individual components and connectors;
- *in-line configuration languages* model configurations explicitly, but specify connector information only as part of the configuration, "in line";
- *explicit configuration languages* model both components and connectors separately from configurations.

The first category, implicit configuration languages, are, by definition given in this paper, *not* ADLs, although they may serve as useful tools in modeling certain aspects of architectures. An example of an implicit configuration language is ArTek [TLPD95]. In ArTek, there is no configuration specification; instead, each connector specifies component ports to which it is attached.

The focus on conceptual architecture and explicit treatment of connectors as first-class entities differentiate ADLs from module interconnection languages (MILs) [DK76, PN86], programming languages, and object-oriented notations and languages (e.g., Unified Method [BR95]). MILs typically describe the *uses* relationships among modules in an *implemented* system and support only one type of connection [AG94a, SG94]. Programming languages describe a system's implementation, whose architecture is typically implicit in subprogram definitions and calls. Explicit treatment of connectors also distinguishes ADLs from OO languages, as demonstrated in [LVM95].

It is important to note, however, that there is less than a firm boundary between ADLs and MILs. Certain ADLs, e.g., Wright and Rapide, model components and connectors at a high level of abstraction and do not assume or prescribe a particular relationship between an architectural description and an implementation. We refer to these languages as being *implementation independent.* On the other hand, several ADLs, e.g., UniCon and MetaH, enforce a high degree of fidelity of an implementation to its architecture. Components modeled in these languages are directly related to their implementations, so that a module interconnection specification may be indistinguishable from an architectural description in such a language. These are *implementation constraining* languages.

2.4. Applications of ADLs

ADLs are special purpose notations whose very specific foci render them suitable for powerful analyses, simulation, and automated code generation. However, they have yet to find their place in mainstream software development. Although current research is under way to bridge the gap that separates ADLs from more widely used design notations [RMRR97], only a small number of existing ADLs have been applied to large-scale, "real-world" examples to date. What these examples do demonstrate is the potential for effective use of ADLs in software projects.

Wright was used to model and analyze the *Runtime Infrastructure* (RTI) of the Department of Defense (DoD) *High-Level Architecture for Simulations* (HLA) [All96]. The original specification for RTI was over 100 pages long. Wright was able to substantially condense the specification and reveal several inconsistencies and weaknesses in it.

SADL was applied to an operational power-control system, used by the Tokyo Electric Power Company. The system was implemented in 200,000 lines of Fortran 77 code. SADL was used to formalize the system's reference architecture and ensure its consistency with the implementation architecture.

Finally, Rapide has been used in several large-scale projects thus far. A representative example is the X/Open Distributed Transaction Processing (DTP) Industry Standard. The documentation for the standard is over 400 pages long. Its reference architecture and subsequent extensions have been successfully specified and simulated in Rapide [LKA+95].

2.5. Architecture vs. Design

Given the above definition of software architectures and ADLs, an issue worth addressing is the relationship between architecture and design. Current literature leaves this question largely unanswered, allowing for several interpretations:

- architecture and design are the same;
- architecture is at a level of abstraction above design, so it is simply another step (artifact) in a software development process; and
- architecture is something new and is somehow different from design (but just how remains unspecified).

All three interpretations are partially correct. To a large extent, architectures serve the same purpose as design. However, their explicit focus on connectors and configurations distinguishes them from traditional software design. At the same time, as a (high level) architecture is refined, connectors lose prominence by becoming distributed across the (lower level) architecture's elements. Such a lower level architecture may indeed be considered to be a design. Keeping this relationship in mind, for reasons of simplicity we will simply refer to architectures as "high level," "low level," and so forth, in the remainder of the paper, while "design" will only refer to the process that results in an architecture.

3. Architectural Domains

ADLs typically share syntactic constructs that enable them to model components and component interfaces, connectors, and configurations.[2] A much greater source of divergence are the different ADLs' conceptual frameworks, and, consequently, their support for modeling architectural semantics. ADL developers typically have decided to focus on a specific aspect of architectures, or an *architectural domain*, which guides their selection of an underlying semantic model and a set of related formal specification notations. These formal notations, in turn, restrict the types of problems for which the ADL is suitable.

This relationship between an architectural domain and candidate formal notations is rarely straightforward or fully understood. In the absence of objective criteria, ADL researchers are forced to base their decisions on intuition, experience, and biases arising from past research accomplishments. Unfortunately, intuition can often be misleading and experience insufficient in a young discipline such as software architectures.

In this paper, we attempt to fill this void. The remainder of this section motivates and formulates a framework for classifying the problems on which architectural models focus (architectural domains), shown in Figure 1. Architectural domains represent broad classes of problems and are likely to be reflected in many ADLs and their associated formal specification language constructs. Their proper understanding is thus necessary. Furthermore, heuristics may be developed over time that will enable easier interchange of architectures modeled in ADLs that focus on particular architectural domains. Finally, such a framework can be used as a guide in developing future ADLs.

2. One can think of these syntactic features as equivalent to a "boxes and arrows" graphical notation with little or no underlying semantics.

```
Representation
Design Process Support
Analysis
   Static
   Dynamic
Evolution
   Specification-Time
   Execution-Time
Refinement
Traceability
Simulation/Executability
```

Figure 1: Architectural domains.

3.1. Representation

A key role of an explicit representation of an architecture is to aid understanding and communication about a software system among different stakeholders. For this reason, it is important that architectural descriptions be simple, understandable, and possibly graphical, with well understood, but not necessarily formally defined, semantics.

Architectural models typically comprise multiple views, e.g., high level graphical view, lower level view with formal specifications of components and connectors, conceptual architecture, one or more implementation architectures, corresponding development process, data or control flow view, and so on. Different stakeholders (e.g., architects, developers, managers, customers) may require different views of the architecture. The customers may be satisfied with a high-level, "boxes and arrows" description, the developers may want detailed component and connector models, while the managers may require a view of the development process.

3.2. Design Process Support

Software architects decompose large, distributed, heterogeneous systems into smaller building blocks. In doing so, they have to consider many issues, make many decisions, and utilize many design techniques, methodologies, and tools.

Modeling architectures from multiple perspectives, discussed in the previous subsection, is only one way of supporting software architects' cognitive processes. Others include delivering design guidance in a timely and understandable fashion, capturing design rationale, and revisiting past design steps.

3.3. Analysis

Architectures are often intended to model large, distributed, concurrent systems. The ability to evaluate the properties of such systems upstream, at the architectural level, can substantially lessen the number of errors passed downstream. Given that unnecessary details are abstracted away in architectures, the analysis task may also be easier to perform than at source code level.

Analysis of architectures may be performed statically, before execution, or dynamically, at runtime. Certain types of analysis can be performed both statically and dynamically.

3.3.1. Static Analysis

Examples of static analysis are internal consistency checks, such as whether appropriate components are connected and their interfaces match, whether connectors enable desired communication, whether constraints are satisfied, and whether the combined semantics of components and connectors result in desired system behavior. Certain concurrent and distributed aspects of an architecture can also be assessed statically, such as the potential for deadlocks and starvation, performance, reliability, security, and so on. Finally, architectures can be statically analyzed for adherence to design heuristics and style rules.

3.3.2. Dynamic Analysis

Examples of dynamic analysis are testing, debugging, assertion checking, and assessment of the performance, reliability, and schedulability of an executing architecture. Saying that an architecture is executing can mean two different things:

- the system built based on the architecture is executing, or
- the runtime behavior of the architecture itself is being simulated.

Clearly, certain analyses, such as performance or reliability, are more meaningful or even only possible in the former case. However, an implementation of the system may not yet exist. Furthermore, it may be substantially less expensive to perform dynamic analyses in the latter case, particularly when the relationship between the architecture and the implemented system is well understood.

3.4. Evolution

Support for software evolution is a key aspect of architecture-based development. Architectures evolve to reflect evolution of a single software system; they also evolve into families of related systems. As design elements, individual components and connectors within an architecture may also evolve. Evolution of components, connectors, and architectures can occur at specification time or execution time.

3.4.1. Specification-Time Evolution

If we consider components and connectors to be types which are instantiated every time they are used in an architecture, their evolution can be viewed simply in terms of subtyping. Since components and connectors are modeled at a high level of abstraction, flexible subtyping methods may be employed. For example, it may be useful to evolve a single component in multiple ways, by using different subtyping mechanisms (e.g., interface, behavior, or a combination of the two) [MORT96].

At the level of architectures, evolution is focused on incremental development and support for system families. Incrementality of an architecture can further be viewed from two different perspectives. One is its ability to accommodate addition of new components and the resulting issues of scale; the other is specification of incomplete architectures.

3.4.2. Execution-Time Evolution

Explicit modeling of architectures is intended to support development and evolution of large and potentially long-running systems. Being able to evolve such systems during execution may thus be desirable and, in some cases, necessary. Architectures exhibit dynamism by allowing replication, insertion, removal, and reconnection of architectural elements or subarchitectures during execution.

Dynamic changes of an architecture may be either planned at architecture specification time or unplanned. Both types of dynamic change must be constrained to ensure that no desired architectural properties are violated.

3.5. Refinement

The most common argument for creating and using formal architectural models is that they are necessary to bridge the gap between informal, "boxes and arrows" diagrams and programming languages, which are deemed too low-level for designing a system. Architectural models may need to be specified at several levels of abstraction for different purposes. For example, a high level specification of the architecture can be used as an understanding and communication tool; a subsequent lower level may be analyzed for consistency of interconnections; an even lower level may be used in a simulation. Therefore, correct and consistent refinement of architectures to subsequently lower levels of abstraction is imperative. Note that, in this sense, code generation is simply a special case of architectural refinement.

3.6. Traceability

As discussed above, a software architecture often consists of multiple views and may be modeled at multiple levels of abstraction (Figure 2). We call a particular view of the architecture at a given level of abstraction (i.e., a single point in the two-dimensional space of Figure 2) an "architectural cross-section." It is critical for changes in one cross-section to be correctly reflected in others. A particular architectural cross-section can be considered "dominant," so that *all* changes to the architecture are made to it and then reflected in others. However, changes will more frequently be made to the most appropriate or convenient cross-section. Traceability support will hence need to exist across all pertinent cross-sections.

One final issue is the consistency of an architecture with system requirements. Changes to the requirements must be appropriately reflected in the architecture; changes to the architecture must be validated against the requirements. Therefore, even though system requirements are in the problem domain, while architecture is in the solution domain, traceability between the two is crucial. For purposes of traceability, requirements can be considered to be at a very high level of architectural abstraction, as shown in Figure 2.

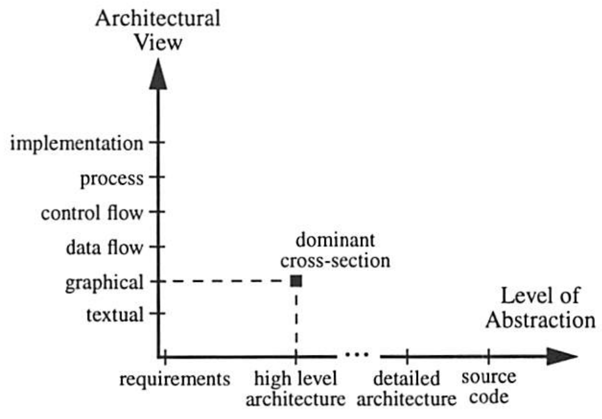


Figure 2: Two-dimensional space of architectural views and levels of abstraction. The vertical axis is a set of discrete values with a nominal ordering. The horizontal axis is a continuum with an ordinal ordering of values, where system requirements are considered to be the highest level of abstraction and source code the lowest. One possible dominant cross-section (graphical view of the high level architecture) is shown.

3.7. Simulation/Executability

Static architectural models are useful for establishing *static* properties of the modeled system. Certain dynamic properties may also be predicted with static models, but only if specific assumptions hold. For example, if the architect can correctly predict execution time and criticality of each component, then schedulability of the encompassing architecture can be evaluated.

On the other hand, other dynamic properties, such as reliability, may by definition require a running system. Also, developers may want to produce an early prototype to, e.g., attempt allocation of architectural elements to components of the physical system. Other stakeholders (e.g., customers or managers) may want to verify early on that the architecture conforms to their wishes. Simulating the dynamic behavior of a high level architecture may thus be preferred to implementing the system: it is a quicker, cheaper, and more flexible way of arriving at the desired information.

A special case of architectural simulation is the execution of the complete implemented system. The ultimate goal of any software design and modeling endeavor is to produce such a system. An elegant and effective architectural model is of limited value, unless it can be converted into a running application. A simulation can only partially depict the final system's dynamic behavior. Manually transforming an architecture into a running system may result in many, already discussed problems of consistency and traceability between the architecture and its implementation. Techniques, such as refinement and traceability discussed above, must be employed to properly accomplish this task.

4. ADL Support for Architectural Domains

In the previous section, we motivated and described different architectural domains in terms of their characteristics and needs of software architectures. Another way of viewing architectural domains is in terms of modeling languages and specific language features needed to support different domains. At the same time, a useful way of understanding and classifying architecture modeling languages is in terms of architectural domains they are intended to support. For these reasons, this section studies the kinds of language facilities that are needed to support each architectural domain, as well as the specific features existing ADLs employ to that end. Our hope is that this discussion will shed light on the relationships among different architectural domains (and their resulting ADL features) and point out both where they can be effectively combined and where we can expect difficulties.

4.1. Representation

Ideally, an ADL should make the structure of a system clear from a configuration specification alone, i.e., without having to study component and connector specifications. Architecture descriptions in *in-line configuration ADLs*, such as Darwin, MetaH, and Rapide tend to be encumbered with connector details, while *explicit configuration ADLs*, such as ACME, Aesop, C2, SADL, UniCon, and Wright have the best potential to facilitate understandability of architectural structure.

One common way of facilitating understandability and communication is by providing a graphical notation, in addition to the textual notation. However, this is only the case if there is a precise relationship between a graphical description and the underlying semantic model. For example, Aesop, C2, Darwin, MetaH, Rapide, and UniCon support such "semantically sound" graphical notations, while ACME, SADL, and Wright do not.

ADLs must also be able to model the architecture from multiple perspectives. As discussed above, several ADLs support at least two views of an architecture: textual and graphical. Each of these ADLs also allows both top-level and detailed views of composite elements. Aesop, MetaH, and UniCon further distinguish different types of components and connectors iconically.

Support for other views is sparse. C2 provides a view of the development process that corresponds to the architecture [RR96]. Darwin's *Software Architect's Assistant* [NKM96] provides a hierarchical view of the architecture which shows all the component types and the "include" relationships among them in a tree structure. Rapide allows visualization of an architecture's execution behavior by building its simulation and animating its execution. Rapide also provides a tool for viewing and filtering events generated by the simulation.

4.2. Design Process Support

As the above examples of C2's, Darwin's, and Rapide's support tools indicate, language features can only go so far in supporting software architects. Adequate tools are also needed. A category of tools that is critical for adequately supporting the design process are *active specification tools*; they can significantly reduce the cognitive load on architects.

Only a handful of existing ADLs provide tools that actively support specification of architectures. In general, such tools can be proactive or reactive. UniCon's graphical editor is proactive. It invokes UniCon's language processing facilities to *prevent* errors during design. Reactive specification tools detect *existing* errors. They may either only inform the architect of the error (*non-intrusive*) or also force the architect to correct it before moving on (*intrusive*). An example of the former is C2's design environment, *Argo*, while MetaH's graphical editor is an example of the latter.

4.3. Analysis

The types of analyses for which an ADL is well suited depend on its underlying semantic model, and to a lesser extent, its specification features. The semantic model will largely influence whether the ADL can be analyzed statically or dynamically, or both. For example, Wright, which is based on communicating sequential processes (CSP) [Hoa85], allows static deadlock analysis of individual connectors and components attached to them. On the other hand, Rapide architectures, which are modeled with partially ordered event sets (posets) [LVB+93], can be analyzed dynamically.

4.3.1. Static Analysis

The most common type of static analysis tools are language parsers and compilers. Parsers analyze architectures for syntactic correctness, while compilers establish semantic correctness. All existing ADLs have parsers. Darwin, MetaH, Rapide, and UniCon also have compilers, which enable these languages to generate executable systems from architectural descriptions. Wright does not have a compiler, but it uses FDR [For92], a model checker, to establish type conformance.

There are numerous other possible types of static analysis of architectures. Several examples are provided by current ADLs. Aesop provides facilities for checking for type consistency, cycles, resource conflicts, and scheduling feasibility in its architectures. C2 uses critics to establish adherence to style rules and design guidelines. MetaH and UniCon both currently support schedulability analysis by specifying non-functional properties, such as criticality and priority. Finally, given two architectures, SADL can establish their relative correctness with respect to a refinement map.

4.3.2. Dynamic Analysis

The ability to analyze an architecture dynamically directly depends on the ADL's ability to model its dynamic behavior. To this end, ADLs can employ specification mechanisms, such as event posets, CHAM, or temporal logic, which can express dynamic properties of a system. Another aspect of dynamic analysis is enforcement of constraints at runtime.

Most existing ADLs tend to view architectures statically, so that current support for dynamic modeling and analysis is scarce. Darwin enables dynamic analysis of architectures by instantiating parameters and components to enact "what if" scenarios. Similarly, Rapide *Poset Browser*'s event filtering features and *Animation Tools* facilitate analysis of architectures through simulation. Rapide's *Constraint Checker* also analyzes the conformance of a Rapide simulation to the formal constraints defined in the architecture. Finally, runtime systems of those ADLs that provide architecture compilation support can be viewed as dynamic analysis facilities.

4.4. Evolution

An architecture can evolve in two different dimensions:

- evolution of individual components and connectors, where the structure of the architecture is not affected, although its behavior may be; and
- evolution of the entire architecture, which affects both the structure and behavior of an architecture.

Evolution in these two dimensions can occur both at architecture specification time and while the architecture is executing.[3]

4.4.1. Specification-Time Evolution

ADLs can support specification-time evolution of individual components and connectors with subtyping. Only a subset of existing ADLs provide such facilities, and even their evolution support is limited and often relies on the chosen implementation (programming) language. The remainder of the ADLs view and model components and connectors as inherently static.

Aesop supports behavior-preserving subtyping of components and connectors to create substyles of a given architectural style. Rapide allows its interface types to inherit from other types by using OO methods, resulting in structural subtyping. ACME also supports structural subtyping via its *extends* feature. C2 provides a more sophisticated subtyping and type checking mechanism. Multiple subtyping relationships among components are allowed: name, interface, behavior, and implementation subtyping, as well as their combinations [MORT96].

Specification-time evolution of complete architectures has two facets: support for incremental development and support for system families. Incrementality of an architecture can be viewed from two different perspectives. One is its ability to accommodate addition of new components to the architecture. In general, *explicit configuration ADLs* can support incremental development more easily and effectively than *in-line configuration ADLs*; ADLs that allow variable numbers of components to communicate through a connector are well suited for incremental development, particularly when faced with unplanned architectural changes [Med97].

Another view of incrementality is an ADL's support for incomplete architectural descriptions. Incomplete architectures are common during design, as some decisions are deferred and others have not yet become relevant. However, most existing ADLs and their supporting toolsets have been built to prevent precisely these kinds of situations. For example, Darwin, MetaH, Rapide, and UniCon compilers, constraint checkers, and runtime systems have been constructed to raise exceptions if such situations arise. In this case, an ADL, such as Wright, which focuses its analyses on information local to a single connector is better suited to accommodate expansion of the architecture than, e.g., SADL, which is very rigorous in its refinement of *entire* architectures.

Still another aspect of static evolution is support for application families. In [MT96], we showed that the number of possible architectures in a component-based style grows exponentially as a result of a linear expansion of a collection of components. All such architectures may not belong to the same logical family. Therefore, relying on component and connector inheritance, subtyping, or other evolution mechanisms is insufficient. An obvious solution, currently adopted only by ACME, is to provide a language construct that allows the architect to specify the family to which the given architecture belongs.

4.4.2. Execution-Time Evolution

There are presently two approaches to supporting evolution of architectures at execution time. The first is what Oreizy calls "constrained dynamism": all runtime changes to the architecture must be known a priori and are specified as part of the architectural model [Ore96].

Two existing ADLs support constrained dynamism. Rapide supports conditional configuration; its *where* clause enables a form of architectural rewiring at runtime, using the *link* and *unlink* operators. Darwin allows runtime replication of components using the *dyn* operator.

The second approach to execution time evolution places no restrictions at architecture specification time on the

3. Saying that an architecture is "executing" can mean either that the architecture is being simulated or that the executable system built based on that architecture is running.

kinds of allowed changes. Instead, the ADL has an architecture modification feature, which allows the architect to specify changes while the architecture is running.

Darwin and C2 are the only ADLs that support such "pure dynamism" [Ore96]. Darwin allows deletion and rebinding of components by interpreting Darwin scripts. C2 specifies a set of operations for insertion, removal, and rewiring of elements in an architecture at runtime [Med96]. C2's *ArchShell* tool enables arbitrary interactive construction, execution, and runtime-modification of C2-style architectures by dynamically loading and linking new architectural elements [Ore96, MOT97]. An issue that needs further exploration is constraining pure dynamic evolution to ensure that the desired properties of architectures are maintained.

4.5. Refinement

ADLs provide architects with expressive and semantically elaborate facilities for specification of architectures. However, an ADL must also enable correct and consistent refinement of architectures to subsequently lower levels of abstraction, and, eventually, to executable systems.

An obvious way in which ADLs can support refinement is by providing patterns, or maps, that, when applied to an architecture, result in a related architecture at a lower level of abstraction. SADL and Rapide are the only two ADLs that provide such support. SADL uses maps to enable correct architecture refinements across styles, while Rapide generates comparative simulations of architectures at different abstraction levels. Both approaches have certain drawbacks, indicating that a hybrid approach may be useful.

Garlan has recently argued that refinement should not be consistent with respect to a single (immutable) law, but rather with respect to particular properties of interest, be they conservative extension (SADL), computational behavior (Rapide), or something entirely different, such as performance [Gar96]. This may be a good starting point towards a successful marriage of the two approaches.

Several ADLs take a different approach to refinement: they enable generation of executable systems directly from architectural specifications. These are typically the *implementation constraining languages*, such as MetaH and UniCon. These ADLs assume the existence of a source file that corresponds to a given architectural element. This approach makes the assumption that the relationship between elements of an architectural description and those of the resulting system will be 1-to-1. Given that architectures are intended to describe systems at a higher level of abstraction than source code modules, this can be considered only a limited form of refinement.

4.6. Traceability

While the problem of refinement essentially focuses only on one axis of Figure 2 (the horizontal axis) and one direction (left to right), traceability may need to cover a large portion of the two-dimensional space and is applicable in both directions. This presents a much more difficult task, indicating why this is the architectural domain in which existing ADLs are most lacking.

The relationships among architectural views (vertical axis) are not always well understood. For example, ADLs commonly provide support for tracing changes between textual and graphical views, such that changes in one view are automatically reflected in the other; however, it may be less clear how the data flow view should affect the process view. In other cases, changes in one view (e.g., process) should never affect another (e.g., control flow). An even bigger hurdle is providing traceability support across *both* architectural views and levels of abstraction simultaneously. Finally, although much research has been directed at methodologies for making the transition from requirements to design (e.g., OO), this process is still an art form. Further research is especially needed to understand the effects of changing requirements on architectures and vice versa.

Traceability is particularly a problem in the way implementation constraining languages approach code generation, discussed in the previous subsection. These ADLs provide no means of guaranteeing that the source modules which are supposed to implement architectural components will do so correctly. Furthermore, even if the specified modules currently implement the needed behavior correctly, there is no guarantee that any future changes to those modules will be traced back to the architecture and vice versa.

4.7. Simulation/Executability

As with dynamic analysis (Section 4.3.2), simulating an architecture will directly depend upon the ADL's ability to model its dynamic behavior. Currently, Rapide is the only ADL that can simulate the architecture itself, by generating event posets. Other ADLs enable generation of running systems corresponding to the architecture.

MetaH and UniCon require preexisting component implementations in Ada and C, respectively, in order to generate applications. Darwin can also construct executable systems in the same manner in C++, and Rapide in C, C++, Ada, VHDL, or its executable sublanguage.

C2 and Aesop provide class hierarchies for their concepts and operations, such as components, connectors, and interconnection and message passing protocols. These hierarchies form a basis from which an implementation of an architecture may be produced. Aesop's hierarchy has been implemented in C++, and C2's in C++, Java, and Ada.

4.8. Summary

Existing ADLs span a broad spectrum in terms of the architectural domains they support. On the one hand, languages like SADL and Wright have very specific, narrow foci. On the other, C2, Rapide, and Darwin support a number of architectural domains. Certain domains, e.g., evolution, refinement, and traceability are only sparsely supported, indicating areas around which future work should be centered. A more complete summary of this section is given in Table 1 below.

Table 1: ADL Support for Architectural Domains

Arch. Domain / ADL	Represent.	Design Process Support	Static Analysis	Dynamic Analysis	Spec-Time Evolution	Exec-Time Evolution	Refinement	Trace.	Simulation/ Executability
ACME	explicit config.; "weblets"	none	parser	none	application families	none	rep-maps across levels	textual <-> graphical	none
Aesop	explicit config.; graphical notation; types distinguished iconically	syntax directed editor; specialized editors for visualization classes	parser; style-specific compiler; type, cycle, resource conflict, and scheduling feasibility checker	none	behavior-preserving subtyping of components and connectors	none	none	textual <-> graphical	*build* tool constructs system glue code in C for pipe-and-filter style
C2	explicit config.; graphical notation; process view; simulation; event filtering	non-intrusive, reactive design critics and to-do lists in *Argo*	parser; critics to establish adherence to style rules and design heuristics	event filtering	multiple subtyping mechanisms; allows partial architectures	pure dynamism: element insertion, removal, and rewiring	none	textual <-> graphical	class framework enables generation of C/C++, Ada, and Java code
Darwin	implicit config.; graphical notation; hierarchical system view	automated addition of ports; propagation of changes across bound ports; property dialogs	parser; compiler	"what if" scenarios by instantiating parameters and dynamic components	none	constrained dynamism: runtime replication of components and conditional configuration	none	textual <-> graphical	compiler generates C++ code
MetaH	implicit config.; graphical notation; types distinguished iconically	intrusive, reactive graphical editor	parser; compiler; schedulability, reliability, and security analysis	none	none	none	none	textual <-> graphical	compiler generates Ada code (C code generation planned)
Rapide	implicit config.; graphical notation; animated simulation; event filtering	none	parser; compiler; constraint checker to ensure valid mappings	event filtering and animation	inheritance (structural subtyping)	constrained dynamism: conditional configuration and dynamic event generation	refinement maps enable comparative simulations of architectures at different levels	textual <-> graphical; constraint checking across refinement levels	simulation by generating event posets; system construction in C/ C++, Ada, VHDL, and Rapide
SADL	explicit config.	none	parser; relative correctness of architectures w.r.t. a refinement map	none	component and connector refinement via pattern maps	none	maps enable correct refinements across levels	refinement across levels	none
UniCon	explicit config.; graphical notation	proactive GUI editor invokes language checker	parser; compiler; schedulability analysis	none	none	none	none	textual <-> graphical	compiler generates C code
Wright	explicit config.	none	parser; model checker for type conformance; deadlock analysis of connectors	none	type conformance for behaviorally related protocols	none	none	none	none

5. Architectural vs. Application Domains

Over the past decade there has been interest in relating architectures, which are in the solution domain, to the problem (or application) domain, leading to the notion of *domain-specific software architectures (DSSAs)* [Tra95]. A DSSA provides a single (generic) *reference architecture*, which reflects the characteristics of a particular problem domain, and which is instantiated for each specific application in that domain. *Architectural styles*, discussed in Section 2, provide another way of relating the problem and solution spaces. Styles are largely orthogonal to DSSAs: a single style may be applicable to multiple application domains; on the other hand, a single DSSA may use multiple styles.

Any attempt to further explore and perhaps generalize the relationship between architectural and application domains would be greatly aided by a classification of application domains. We are unaware of any such classification, although Jackson identified a number of *domain characteristics* that could serve as a starting point for one [Jac95]:

- *static* vs. *dynamic* domains, with the latter being application domains having an element of time, events, and/or state;
- *one-dimensional* vs. *multi-dimensional* domains;
- *tangible* vs. *intangible* domains, with the latter typically involving machine representations of abstractions (such as user interfaces);
- *inert* vs. *reactive* vs. *active* dynamic domains; and
- *autonomous* vs. *programmable* vs. *biddable* active dynamic domains.

Given these application domain characteristics, one can easily identify a number of useful relationships with architectural domains. For instance, support for evolution, executability and dynamic analysis are more important for dynamic domains than for static domains. As another example, reactive domains are naturally supported by a style of representation (e.g., Statecharts [Har87]) that is different from that in active domains (e.g., CHAM [IW95]). As we deepen our understanding of architectural domains, we will be able to solidify our understanding of their relationship with application domains.

6. Conclusions

Software architecture research has been moving forward rapidly. A number of ADLs and their supporting toolsets have been developed; many existing styles have been adopted and new ones invented. Theoretical underpinnings for the study of software architectures have also begun to emerge in the form of definitions [PW92, GS93] and formal classifications of styles [SC96] and ADLs [Med97, MT97].

This body of work reflects a wide spectrum of views on what architecture is, what aspects of it should be modeled and how, and what its relationship is to other software development concepts and artifacts. This divergence of views has also resulted in a divergence of ADLs' conceptual frameworks (as defined in Section 2). Such fragmentation has made it difficult to establish whether there exists in ADLs a notion similar to computational equivalence in programming languages. Furthermore, sharing support tools has been difficult.

ACME has attempted to provide a basis for interchanging architectural descriptions across ADLs. However, ACME has thus far been much more successful at achieving architectural interchange at the syntactic (i.e., structural) level, than at the semantic level. Although some of the ACME team's recent work looks encouraging, this still remains an open problem. One of the reasons ACME has encountered difficulties is precisely the fact that there is only limited agreement in the architecture community on some fundamental issues, the most critical of which is what problems architectures should attempt to solve.

This paper presents an important first step towards a solution to this problem. We have recognized that the field of software architecture is concerned with several domains and that every ADL reflects the properties of one or more domains from this set. Architectural domains thus provide a unifying view to what had seemed like a disparate collection of approaches, notations, techniques, and tools. The task of architectural interchange can be greatly aided by studying the interrelationships among architectural domains. Existing ADLs can be better understood in this new light and new ADLs more easily developed to solve a specific set of problems.

Much further work is still needed, however. Our current understanding of the relationship between architectural domains and formal semantic theories (Section 2) is limited. Also, we need to examine whether there exist techniques that can more effectively support the needs of particular architectural domains than those provided by existing ADLs. Finally, a more thorough understanding of the relationship between architectural and application domains is crucial if architecture-based development is to fulfill its potential.

7. Acknowledgements

We would like to thank Richard Taylor, Peyman Oreizy, Jason Robbins, David Redmiles, and David Hilbert for their participation in numerous discussions of issues concerning ADLs. We also thank the DSL reviewers for their helpful reviews.

Effort partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0218 and F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Approved for Public Release — Distribution Unlimited.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973.

8. References

[AG94a] R. Allen and D. Garlan. Formal Connectors. Technical Report, CMU-CS-94-115, Carnegie Mellon University, March 1994.

[AG94b] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71-80, Sorrento, Italy, May 1994.

[All96] R. Allen. HLA: A Standards Effort as Architectural Style. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 130-133, San Francisco, CA, October 1996.

[BR95] G. Booch and J. Rumbaugh. *Unified Method for Object-Oriented Development.* Rational Software Corporation, 1995.

[DK76] F. DeRemer and H. H. Kron. Programming-in-the-large versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, pages 80-86, June 1976.

[For92] *Failures Divergence Refinement: User Manual and Tutorial.* Formal Systems (Europe) Ltd., Oxford, England, October 1992.

[GAO94] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

[Gar95] D. Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.

[Gar96] D. Garlan. Style-Based Refinement for Software Architecture. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 72-75, San Francisco, CA, October 1996.

[GMW95] D. Garlan, R. Monroe, and D. Wile. ACME: An Architectural Interconnection Language. Technical Report, CMU-CS-95-219, Carnegie Mellon University, November 1995.

[GMW97] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Interchange Language. Submitted for publication, January 1997.

[GPT95] D. Garlan, F. N. Paulisch, and W. F. Tichy, editors. *Summary of the Dagstuhl Workshop on Software Architecture*, February 1995. Reprinted in ACM Software Engineering Notes, pages 63-83, July 1995.

[GS93] D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-99. SRI International, 1988

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[IW95] P. Inverardi and A. L. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, pages 373-386, April 1995.

[Jac95] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, pages 336-355, April 1995.

[Luc87] D. Luckham. *ANNA, a language for annotating Ada programs: reference manual*, volume 260 of

Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1987.

[LV95] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.

[LVB+93] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems. *Journal of Systems and Software*, pages 253-265, June 1993.

[LVM95] D. C. Luckham, J. Vera, and S. Meldal. Three Concepts of System Architecture. Unpublished Manuscript, July 1995.

[Med96] N. Medvidovic. ADLs and Dynamic Architecture Changes. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 24-27, San Francisco, CA, October 1996.

[Med97] N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report, UCI-ICS-97-02, University of California, Irvine, January 1997.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, Barcelona, September 1995.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3-14, San Francisco, CA, October 1996.

[MOT97] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pages 190-198, Boston, MA, May 17-19, 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pages 692-700, Boston, MA, May 17-23, 1997.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 24-32, San Francisco, CA, October 1996.

[MQR95] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pages 356-372, April 1995.

[MT96] N. Medvidovic and R. N. Taylor. Reusing Off-the-Shelf Components to Develop a Family of Applications in the C2 Architectural Style. In *Proceedings of the International Workshop on Development and Evolution of Software Architectures for Product Families*, Las Navas del Marqués, Ávila, Spain, November 1996.

[MT97] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. To appear in *Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 22-25, 1997.

[MTW96] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28-40, Los Angeles, CA, April 1996.

[NKM96] K. Ng, J. Kramer, and J. Magee. A CASE Tool for Software Architecture Design. *Journal of Automated Software Engineering (JASE), Special Issue on CASE-95*, 1996.

[Ore96] Peyman Oreizy. Issues in the Runtime Modification of Software Architectures. Technical Report, UCI-ICS-96-35, University of California, Irvine, August 1996.

[Pet62] C. A. Petri. Kommunikationen Mit Automaten. PhD Thesis, University of Bonn, 1962. English translation: Technical Report RADC-TR-65-377, Vol.1, Suppl 1, Applied Data Research, Princeton, N.J.

[PN86] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, pages 307-334, October 1989.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pages 40-52, October 1992.

[RMRR97] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. Technical Report, UCI-ICS-97-35, University of California, Irvine, August 1997.

[RR96] J. E. Robbins and D. Redmiles. Software architecture design from the perspective of human cognitive needs. In *Proceedings of the California Software Symposium (CSS'96)*, Los Angeles, CA, USA, April 1996.

[SC96] M. Shaw and P. Clements. Toward Boxology: Preliminary Classification of Architectural Styles. In A. L. Wolf, ed., *Proceedings of the Second*

International Software Architecture Workshop (ISAW-2), pages 50-54, San Francisco, CA, October 1996.

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, pages 314-335, April 1995.

[SG94] M. Shaw and D. Garlan. Characteristics of Higher-Level Languages for Software Architecture. Technical Report, CMU-CS-94-210, Carnegie Mellon University, December 1994.

[Spi89] J. M. Spivey. *The Z notation: a reference manual.* Prentice Hall, New York, 1989.

[TLPD95] A. Terry, R. London, G. Papanagopoulos, and M. Devito. The ARDEC/Teknowledge Architecture Description Language (ArTek), Version 4.0. Technical Report, Teknowledge Federal Systems, Inc. and U.S. Army Armament Research, Development, and Engineering Center, July 1995.

[Tra95] W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, July 1995.

[Ves93] S. Vestal. A Cursory Overview and Comparison of Four Architecture Description Languages. Technical Report, Honeywell Technology Center, February 1993.

[Ves96] S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.

[Wolf96] A. L. Wolf, editor. *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, October 1996.

The Zephyr Abstract Syntax Description Language

Daniel C. Wang Andrew W. Appel Jeff L. Korn
Christopher S. Serra
Department of Computer Science, Princeton University, Princeton, NJ, 08544
{danwang,appel,jlk}@cs.princeton.edu, csserra@cs.wisc.edu

Abstract

The Zephyr[1] Abstract Syntax Description Language (ASDL) describes the abstract syntax of compiler intermediate representations (IRs) and other tree-like data structures. Just as the lexical and syntactic structures of programming languages are described with regular expressions and context free grammars, ASDL provides a concise notation for describing the abstract syntax of programming languages. Tools can convert ASDL descriptions into the appropriate data-structure definitions and functions to convert the data-structures to or from a standard flattened representation. This makes it easier to build compiler components that interoperate.

Although ASDL lacks subtyping and inheritance, it is able to describe the Stanford University Intermediate Format (SUIF) compiler IR, originally implemented in C++. We have built a tool that converts ASDL into C, C++, Java, and ML data-structure definitions and conversion functions. We have also built a graphical browser-editor of ASDL data structures. ASDL shares features found in many network interface description languages (IDLs), algebraic data types, and languages such as ASN.1 and SGML. Compared to other alternatives ASDL is simple and powerful. This document describes ASDL in detail and presents an initial evaluation of ASDL.

1 Introduction

Reusable components make it easy to build compilers to test new compilation techniques. The components of a compiler communicate with each other through an intermediate representation (IR), which is a description of a program suitable for optimization and analysis. If compiler components can exchange compatible IRs they can interoperate.

To interoperate, components need an implementation of an IR and a way to transmit IR values to other components. A simple way to transmit IR values across different components is to read and write the IR to a file in a standard format. These files are called pickles, and conversion to pickles is called pickling or marshaling [BNOW93]. Since different compiler research groups program in different implementation languages, IRs need to be implemented in more than one programming language. Otherwise compiler components written in different languages cannot interoperate.

Unfortunately, many IRs are only described by one implementation in one language. For these IRs it can be hard to separate the abstract structure of the IR from implementation artifacts. Since IRs are often recursively defined tree-like data structures, once the IR is understood it is easy but tedious to develop different implementations in other languages. Writing functions to pickle the IR is also easy but tedious.

A parser implementation is a poor way to describe the concrete syntax of a programming language. A set of data structures is a poor way to describe an IR. This document describes the Abstract Syntax Description Language (ASDL), a simple declarative language for describing the abstract structures of IRs. IRs described with ASDL are converted into an implementation automatically by tools. Tools generate the data-structure definitions for a target language as well as the pickling functions and other supporting code. ASDL is designed so that it is easy to convert descriptions into readable implementations. ASDL descriptions are more concise than data-structure definitions in languages such as C, C++, and Java.

The idea of special notation for tree-like data structures is not new. Compiler-construction systems and attribute-evaluation tools contain small sublanguages that are descriptions of tree-like data structures [GHL+92, Bat96, Vol91, JPJ+90]. Programming languages that provide support for algebraic data types also have concise notation for defining tree-like data struc-

[1] Zephyr is a compiler infrastructure project conducted jointly at the University of Virginia and Princeton University (see `http://www.cs.virginia.edu/zephyr`). This work is supported in part by the Department of Defense under contract MDA904-97-C-0247 and DARPA order number E381, and the National Science Foundation grant ASC-9612756.

	Good Notation	Language Independent	Simple
Compiler Construction Systems	yes	no	yes
Algebraic Data Types	yes	no	yes
ASN.1	could be better	yes	no
SGML	no	yes	no
Network IDLs	no	yes	yes

Table 1: Evaluation of existing systems

tures [BMS80]. Unfortunately these systems and languages do not solve the problem of providing IR implementations for more than one programming language.

Languages like SGML [GR90] and ASN.1 [ISO87, ITU95b] are not much more than complex description languages for tree-like data structures. These descriptions are declarative specifications of structured data, independent of a particular implementation language. However, these languages have many features not needed in the description of compiler IRs. For example, ASN.1 contains thirteen different primitive string types. SGML has a "tag minimization" feature to help define formats that are easier to write by humans but more difficult to parse.

Although SGML and ASN.1 do solve the problems of component interoperation, they seem verbose, cryptic, and complex. The extra complexity of these systems alone makes them unsuitable, since the resources spent understanding and using the systems can often be greater than the time the systems saves the programmer. There already are groups advocating the use of a simplified subset of SGML for for distribution of web content [XML97]. ASDL in some ways can be viewed as simplification of ASN.1.

Heterogeneous networked systems have solved a similar component interoperation problem with interface description languages (IDLs) that describe abstract interfaces to network services. Tools automatically generate glue code from an IDL description and export a service to the network. ONC RPC [Sri95], OMG CORBA [Obj95], and Xerox's ILU [Xer96] are examples of this approach. Unfortunately CORBA and the other IDLs have awkward encodings for the tree-like data structures seen in IRs.

Ideally those in the compiler research community could reuse existing solutions. Unfortunately none of the existing systems are a good solution to the problem of building interoperable compiler components and concisely defining abstract IRs. Table 1 summarizes our evaluation of existing systems. The following is a summary of the concrete design goals of ASDL.

- The language must be simple and concise.
- The language must be able to encode existing IR's such as SUIF [W$^+$94], FLINT [Sha97], and lcc's IR [FH95].
- Tools that use the language must initially be able to produce code for C, C++, Java, and ML.
- Tools must be able to produce code designed to be understood by programmers, not just other tools.
- Language features must have a natural encoding in all the target languages.

The next sections present the language informally, evaluate the language, and discuss related and future work.

2 ASDL by Example

$$
\begin{array}{rcl}
\textit{definitions} & = & \{\textit{typ\_id}\ \texttt{"="}\ \textit{type}\} \\
\textit{type} & = & \textit{sum\_type}\ |\ \textit{product\_type} \\
\textit{product\_type} & = & \textit{fields} \\
\textit{sum\_type} & = & \textit{constructor}\ \{\texttt{"|"}\ \textit{constructor}\} \\
 & & [\texttt{"attributes"}\ \textit{fields}] \\
\textit{constructor} & = & \textit{con\_id}\ [\textit{fields}] \\
\textit{fields} & = & \texttt{"("}\ \{\textit{field}\ \texttt{","}\}\ \textit{field}\ \texttt{")"} \\
\textit{field} & = & \textit{typ\_id}\ [\texttt{"?"}\ \ |\ \texttt{"*"}]\ [\textit{id}]
\end{array}
$$

Figure 1: Grammar of ASDL. Braces indicate zero or more. Brackets indicate zero or one.

Figure 1 is the grammar for ASDL. The syntax of ASDL has been designed to be natural and intuitively obvious to anyone familiar with context free grammars (CFG) or algebraic data types. In the same way that an unambiguous CFG can be viewed as describing the structure of parse trees, ASDL describes the structure of tree-like data structures. An ASDL description consists of a sequence of productions, which define the structure of a tree-like data structure. ASDL descriptions are tree grammars.

ASDL is simple enough to describe with a few examples. Since one goal is to generate human readable code, there are a few restrictions on ASDL whose rationale is not completely obvious. These restrictions and their motivation are described as they arise. Figure 3 is the ASDL description of a trivial programming language.

2.1 Lexical Issues

$$\begin{array}{rcl}
\mathit{upper} & = & \texttt{"A"} \mid \ldots \mid \texttt{"Z"} \\
\mathit{lower} & = & \texttt{"a"} \mid \ldots \mid \texttt{"z"} \\
\mathit{alpha} & = & \texttt{"\_"} \mid \mathit{upper} \mid \mathit{lower} \\
\mathit{alpha\_num} & = & \mathit{alpha} \mid \texttt{"0"} \mid \ldots \mid \texttt{"9"} \\
\mathit{typ\_id} & = & \mathit{lower}\ \{\mathit{alpha\_num}\} \\
\mathit{con\_id} & = & \mathit{upper}\ \{\mathit{alpha\_num}\} \\
\mathit{id} & = & \mathit{typ\_id} \mid \mathit{con\_id}
\end{array}$$

Figure 2: Lexical structure

Figure 2 is a description of the lexical structure of tokens used in the ASDL grammar in Figure 1. The names of constructors and types in the description contain informal semantic information that should be preserved by a tool when translating descriptions into implementations. To keep the mapping from ASDL names to target language names simple, the names of types and constructors are restricted to the intersection of valid identifiers in the initial set of target languages. To help the reader distinguish between types and constructor names, types are required to begin with a lower case letter and constructor names must begin with an upper case letter. Rather than restricting ASDL names to exclude the union of keywords in all target language, ASDL tools will have to keep track and correct conflicts between target language keywords and the type and constructor names.

ASN.1 has a similar restrictions. However, the ASN.1 equivalent of ASDL types must begin with an upper case letter, and non-type identifiers must begin with a lower case letter. The ASN.1 restrictions are incompatible with many common stylistic conventions in ML, Java, C++, and C. For example, enumerated constants in ASN.1 must begin with a lower case letter, but C style languages conventionally use all uppercase identifiers for enumerated constants.

2.2 ASDL Fundamentals

An ASDL description consists of three fundamental constructs: types, constructors, and productions. A type is defined by productions that enumerate the constructors for that type. In Figure 3 the first production describes a *stm* type. A value of the *stm* type is created by one of three different constructors **Compound**, **Assign**, and **Print**. Each of these constructors has a sequence of

$$\begin{array}{rcl}
\mathit{stm} & = & \mathbf{Compound}(\mathit{stm}, \mathit{stm}) \\
 & \mid & \mathbf{Assign}(\mathit{identifier}, \mathit{exp}) \\
 & \mid & \mathbf{Print}(\mathit{exp\_list}) \\
\mathit{exp\_list} & = & \mathbf{ExpList}(\mathit{exp}, \mathit{exp\_list}) \mid \mathbf{Nil} \\
\mathit{exp} & = & \mathbf{Id}(\mathit{identifier}) \\
 & \mid & \mathbf{Num}(\mathit{int}) \\
 & \mid & \mathbf{Op}(\mathit{exp}, \mathit{binop}, \mathit{exp}) \\
\mathit{binop} & = & \mathbf{Plus} \mid \mathbf{Minus} \mid \mathbf{Times} \mid \mathbf{Div}
\end{array}$$

Figure 3: Simple ASDL description

fields that describe the type of values associated with a constructor.

The **Compound** constructor has two fields whose values are of type *stm*. One can interpret the production as defining the structure of *stm* trees which can have three different kinds of nodes **Compound**, **Assign**, and **Print** where the **Compound** node has two children that are subtrees that have the structure of a *stm* tree.

Notice that the *binop* type consists of only constructors which have no fields. Types like *binop* are therefore finite enumerations of values. Tools can easily recognize this and represent these types as enumerations in the target language. ASDL does not provide an explicit enumeration type, unlike ASN.1 and the various IDLs. Tools should recognize this idiom and use an appropriate encoding.

There are three primitive pre-defined types in ASDL. Figure 3 uses two of them *int* and *identifier*. The *int* type represents signed integers of infinite precision. Specific tools may choose to produce language interfaces that represent them as integers of finite precision. These language interfaces should appropriately signal an error when they are unable to represent such a value during unpickling. The *identifier* type is analogous to Lisp symbols. ASDL also provides a primitive *string* type.

2.3 Generating Code from ASDL Descriptions

From the definitions in Figure 3, it is easy to automatically generate data type declarations in target languages such as C, C++, Java, and ML. For languages like C, each type is represented as a tagged union of values. Languages like Java and C++ have a single abstract base class for each type and concrete subclasses of the base class for each variant of the type.

Figure 4 shows one way to translate the *stm* type into C. Each ASDL type is represented as a pointer to a structure. The structure contains a "kind" tag that indicates which variant of the union the current value holds. It is also convenient to have functions that allocate space and properly initialize the different variants of *stm*. Notice that the *binop* is translated as an enumeration.

```
typedef struct _stm *stm_ty;
struct _stm {
   enum {Compound_kind=1, Assign_kind=2,
          Print_kind=3} kind;
   union {
      struct { stm_ty stm1; stm_ty stm2; } Compound;
      struct { ... } Assign;
      struct { ... } Print;
   } v;
};
...
enum binop_ty {Plus=1, Minus=2, Times=3, Div=4};
...
stm_ty Compound (stm_ty stm1, stm_ty stm2) {
   stm_ty p;

   p = malloc(sizeof(*p));
   p->kind = Compound_kind;
   p->v.Compound.stm1 = stm1;
   p->v.Compound.stm2 = stm2;
   return p;
}
stm_ty Assign (identifier_ty identifier1, exp_ty exp1) { ... }
stm_ty Print (exp_list_ty exp_list1) { ... }
...
```

Figure 4: Simple translation to C

```
abstract public class stm {
   protected int _k;
   public final int kind(){ return _k; }
   public static final int Compound = 1;
   public static final int Assign = 2;
   public static final int Print = 3;
}

public class Compound extends stm {
   public stm stm1; public stm stm2;
   public Compound(stm stm1, stm stm2) { ... }
}

public class Assign extends stm { ... }
public class Print extends stm { ... }
...
```

Figure 5: Simple translation to Java

```
datatype stm = Compound of (stm * stm)
             | Assign of (identifier * exp)
             | Print of (exp_list)
             ...
```

Figure 6: Simple translation to ML

Figure 5 shows one possible encoding in Java, which is applicable for many other languages with objects, such as C++ and Modula-3. The *stm* type is represented as an abstract base class. The various ASDL constructors are translated into subclasses of the abstract base class. Each constructor class inherits a tag that identifies which variant of *stm* it is. The translation to a language like ML that has algebraic data types is almost trivial (See Figure 6).

Our prototype definitions generator tool uses these encoding schemes to automatically translate ASDL into C, C++, Java, and ML. These encodings are simple and uniform; they are not necessarily the most efficient possible. Better tools can potentially generate more efficient encodings, or allow the programmer to specify an encoding explicitly.

2.4 Field Names

Since languages like C, Java, and C++ access components of aggregates with named fields, ASDL descriptions allow the specification of a field name to access the values of constructor fields. In the absence of a supplied field name tools can easily create field names based on the position and type of a constructor field. Since field names often encode semantic information, the ability to provide names for fields in the descriptions improves the readability of descriptions and the code generated from those descriptions. There are no restrictions on the case of the first character of field names. Figure 7 contains a fragment of the original description which also includes field names.

2.5 Sequences

The *exp_list* type illustrates a common idiom for expressing a uniform sequence of some type. Sequences of a uniform type occur throughout descriptions and general programming. ASDL provides special support for sequences of values through the "*" (sequence) qualifier, which means that the type of some value is an sequence of zero or more elements of that type. Figure 8 demonstrates its use in the context of the previous definitions. The sequence qualifier is not just syntactic sugar. It provides a mechanism in the description for the writer to more clearly specify the intent giving tools that generate code more freedom to use appropriate representations in the native language. For example, a tool may translate a

```
stm      =  Compound(stm head, stm next)
         |  Assign(identifier lval, exp rval)
         |  Print(exp_list args)
exp_list =  ExpList(exp head, exp_list next)
         |  Nil
   ...
```

Figure 7: ASDL description with named fields

```
stm   =  Compound(stm head, stm next)
      |  Assign(identifier lval, exp rval)
      |  Print(exp* args)
  ...
```

Figure 8: ASDL description with sequences

sequence type into an array or another built-in sequence type that the target language supports, such as a polymorphic or templated list type.

ASN.1, SGML, ONC RPC, OMG IDL, and Xerox's ILU have qualifiers for sequence types. These systems, except for ASN.1 and SGML, also have qualifiers to specify the minimum or maximum length of a sequence. ASN.1 and SGML also have qualifiers to specify that the order of components in sequences has no meaning. They support the notion of a set of values. ASDL does not support this feature, since sequences can model sets.

2.6 Product Types, Attributes, and Options

Those familiar with EBNF[Wir77] or algebraic data types may expect to be able to write descriptions with productions such as

```
t  =  C(int, (int, int)*) .
```

However, complex expressions of this type are not allowed in ASDL. The reason behind this restriction is that not all the source languages support a natural encoding for complex type expressions. One would expect that equivalent type expressions are translated into compatible types in the target language. Since the semantics of aggregate types in C and C++ require each new aggregate (struct/class) definition to be a new distinct type, tools would have to use target language type abbreviation mechanisms (e.g. typedef) to achieve this effect. So a tool must assign a name to the type that a programmer must use and remember. There are several obvious ways to automatically generate type names for the expressions, however it would be preferable to require description writers to provide semantically meaningful names to these intermediate types, since generated code is intended to be readable by the programmer. So the above would be written as

```
       t  =  C(int, int_pair*)
int_pair  =  IP(int, int) .
```

This restriction is unsatisfactory since it requires descriptions writers to also provide a name for the single constructor, **IP**, of this type. To overcome this problem, ASDL provides (Cartesian) product types which are productions that define a type that is an aggregate of several values of different types. Product types are also restricted in that they can not lead to recursive definitions, since recursive product type definitions to not describe tree structures. Another way to encode the first expression in ASDL which avoids the extra constructor would be

```
       t  =  C(int, int_pair*)
int_pair  =  (int, int) .
```

Often several constructors of a type share a set of common values. To make this explicit, ASDL includes an attribute notation. Since all variants of a type carry the values of an attribute, its fields can be accessed without having to discriminate between the various constructors. Attributes can be seen as providing some limited features of inheritance.

Most languages provide the notion of a special distinguished empty value (NULL, nil, NONE). ASDL provides a convention for specifying that certain values may be empty with the "?" (optional) qualifier.

```
  pos  =  (string? file, int line, int offset)
  stm  =  Compound(stm head, stm next)
       |  Assign(identifier lval, exp rval)
       |  Print(exp* args)
          attributes (pos p)
 real  =  (int mantissa, int exp)
  exp  =  Id(identifier)
       |  Num(int)
       |  Op(exp, binop, exp)
          attributes (real? value)
binop  =  Plus | Minus | Times | Div
```

Figure 9: ASDL description with products, attributes, and options

Figure 9 is an example of an ASDL description that uses, products, attributes, and options. It is important to emphasize that ASDL says nothing about how a definition should be translated by a tool into a specific concrete implementation. The description language and external data encoding are fixed; the particular target language interfaces are not. Different tools may produce different language interfaces, as long as the pickle formats used by various tools are compatible. For example since ASDL does not provide a primitive type for real numbers the ASDL description in Figure 9 describes a real type in terms of two arbitrary precision integers. Programmers can provide the translation tool hints and conversion functions so actual implementations of the above IR use native floating-point values.

ASDL product types are nothing more than records. Many IDLs (XDR, ISL, and IDL) which have support for records place a similar restriction on the recursive definitions of structures. The description language for Xerox's ILU system (ISL) forbids complex type expressions in the same way ASDL does. The XDR specification allows for complex type expressions, but common implementations of the tools do not allow the use of them.[2] ASN.1, SGML, and OMG's IDL allow the construction of complex type expressions. All the previously mentioned languages have a similar optional qualifier. Although attributes can be simulated in all the description languages, only SGML has a notion of attributes similar to ASDL.

2.7 Pickles

```
...
void pkl_write_exp(...) {...}
void pkl_write_exp_list(...) {...}
void pkl_write_binop(...) {...}
void pkl_write_stm(stm_ty x, outstream_ty s) {
  switch(x->kind) {
    case Compound_kind:
      pkl_write_int(1, s);
      pkl_write_stm(x->v.Compound.stm1, s);
      pkl_write_stm(x->v.Compound.stm2, s);
      break;
    case Assign_kind:
      pkl_write_int(2, s);
      ...
      break;
    case Print_kind:
      pkl_write_int(3, s);
      ...
      break;
    default: pkl_die();
  }
}
...
exp_ty pkl_read_exp(instream_ty s) {...}
...
stm_ty pkl_read_stm(instream_ty s) {...}
```

Figure 10: Automatically Generated Pickler

Since ASDL data structures have a tree-like form, they can be represented linearly with a simple prefix encoding. It is easy to generate functions that convert to and from the linear form. Figure 10 is a generated routine that "pickles" the *stm* type seen previously in Figure 4. A pre-order walk of the data structure is sufficient to convert a *stm* to its pickled form. The walk is implemented as recursively defined functions for each type in an ASDL definition. Each function visits a node of that type and recursively walks the rest of the tree.

In Figure 10 the function `pkl_write_stm` dispatches based on the kind of *stm* constructor of the node being visited. It visits the node by writing a unique tag to identify the constructor to an output stream and then recursively visits any values carried by the constructor. Tags are assigned based on the order of constructor definition in the description. Values are visited from left to right based of the order in the definition. If there are any attribute values associated with a type, they are visited in left to right order after writing the tag but before visiting the values unique to a given constructor. In this case there are no attribute values. Since the prefix encoding does not represent pointers in the data structure the linear form is significantly smaller than the pointer data structures.

The function `pkl_write_stm` calls `pkl_write_int` to output integer values to the output stream. Since ASDL integers are intended to be of infinite precision they are represented with a variable-length, signed-magnitude encoding. If most integer values tend to be values near zero, this encoding of integers may use less space than a fixed precision representation.

Sequence types are represented with an integer length-header followed by that many values. Optional values are preceded by an integer header that is either one or zero. A zero indicates that the value is empty (NONE, nil, or NULL) and no more data follows. A one indicates that the next value is the value of the optional value. Identifiers and strings are encoded with an integer size-header followed by the raw bytes needed to reconstruct the string or identifier. All the headers are encoded with the same arbitrary precision integer encoding described previously.

Product types are written out sequentially without any tag. The ASDL pickle format requires that both the reader and writer of the pickler agree on the type of the pickle. Other than constructor tags there is no explicit type information in the pickle. The prefix encoding of trees, variable-length integer encoding, and lack of explicit type information, all help keep the size of pickles small. Smaller pickles reduce the system IO since there is less data to write or read. Smaller pickles are also more likely to fit completely in the cache of the IO system.

[2] rpcgen on Solaris and OSF 3.2

The ASDL pickle format resembles the Packed Encoding Rules (PER) of ASN.1 [ITU95a]. Like the ASDL pickle format the PER is a prefix encoding of tree values. Neither format encodes redundant type information. Rather than using a variable-precision encoding for integer values and headers, the PER determines from the ASN.1 specification the maximum precision need for a particular value and uses fixed precision integers to represent those values. In the case where an ASN.1 specification does not constrain an integer value so the maximum precision can be determined, ASN.1 resorts to a variable precision integer encoding. The PER encoding of optional values is also slightly different from the ASDL approach. PER optional values are encoded as a bitmap that precedes a record of values that may contain optional values.

Preliminary performance evaluations of the generated pickled code suggest that they are efficient enough not to be the primary performance bottlenecks. Writing pickled values is dominated by IO time, while reading values is dominated by memory allocator time.

3 Evaluation

The next few sections describe insights gained by attempting to respecify an existing compiler IR in ASDL, an evaluation of ASDL's syntax, and some initial experiences using ASDL related tools to build applications.

3.1 ASDL SUIF

ASDL has been used to respecify the core IR of an existing compiler infrastructure, the Stanford University Intermediate Format [W$^+$94] (SUIF) written in C++. Being able to specify existing compiler IRs in ASDL is one of the key design goals of ASDL. SUIF uses an object oriented framework to implement its core IR.

$$\begin{array}{rcl}
\textit{instruction} & = & \textbf{In\_rrr}(\ldots) \\
 & | & \textbf{In\_ldc}(\ldots) \\
 & & \ldots \\
 & | & \textbf{In\_gen}(\ldots) \\
 & & \mathsf{attributes}\ (\ldots)
\end{array}$$

Figure 12: ASDL encoding of SUIF class hierarchy

Figure 11 shows the class hierarchy for SUIF. Looking at classes such as *sym_node* and *instruction*, it is easy to model these classes as types in ASDL with their subclasses represented as constructors in ASDL. Fields that the subclasses may inherit from *instruction* are modeled as common fields and use the attribute mechanism in ASDL. Figure 12 outlines this approach for the *instruction* class.

There are situations where the intent of C++ SUIF code does not fit well into the ASDL model, but these situations are isolated. For example *proc_symtab*, *tree_proc*, and *enum_type* require us to simulate two levels of inheritance in the ASDL description. Attributes in ASDL provide only one level of inheritance. To handle cases where class hierarchies have more than one level of inheritance, extra intermediate types have to be introduced, making the ASDL description less than perfect.

Figure 13 demonstrates how to simulate two levels of inheritance in ASDL. The class *tree_proc* inherits from *tree_block*. The class *tree_block* inherits from *tree_node*. The ASDL description models this by introducing a new intermediate type *tree_block_rest* which consists of two constructors. The **Tree_block** constructor has all the fields from the *tree_block* class. The **Tree_proc** constructor contains or all the fields inherited from the *tree_block* class and the fields of *tree_proc* class. There is a slightly better encoding that uses attributes, since the **Tree_block** and **Tree_proc** constructors share a common set of fields.

The C++ code also uses subtyping to express the constraint that a field must be a particular subtype of an abstract class. In ASDL this is equivalent to using an ASDL constructor as a type in the description. This problem can be solved by allowing the ASDL encoding to be more permissive by not encoding this constraint. Figure 14 provides an example. The `pr` field is declared as a *tree_proc* class, which is a subtype of *tree_node*. In the ASDL description the *tree_proc* corresponds to a constructor, not a type, so `pr` in the ASDL description cannot be declared with the proper type. Instead the ASDL description must use the *tree_node* type.

There are issues not unique to C++, such as how to encode pointers to other tree nodes, making the data structures arbitrary graphs, or the encoding of pointers to other external data structures such as symbol tables. These issues can easily be handled by including unique identifiers and an auxiliary mapping from identifiers to values, to simulate the effect of pointers.

Although the ASDL description is not a verbatim translation of the C++ implementation, the majority of the ASDL description captures most of the features of the C++ implementation in a natural way. The ASDL encoding is less restrictive than the C++ implementation, so functions can be written that convert between data structures that use the original C++ implementation of SUIF and the equivalent ASDL data structures and without loss of information. Using these functions along with code automatically generated from the ASDL description, we have built a tool tool that allows the compilers written in ML and Java to interface to the existing

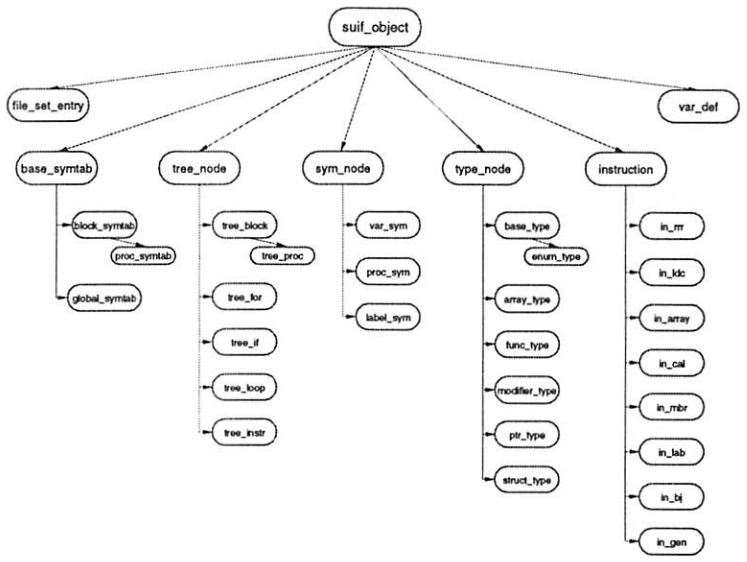


Figure 11: SUIF class hierarchy

SUIF compiler components[Ser97].

3.2 ASDL Syntax

Table 2 compares the size of ASDL SUIF description and the C++ implementation. The C++ kernel is the core set of source files that defines the structure of the SUIF IR and the related support functions. From the kernel there are ten core header files that describe IR structure. The ASDL description was written by examining the original C++ sources. The ASDL description uses the same set of identifiers that the original C++ code uses. The ASDL description does not completely capture all aspects of the C++ code as, explained previously. Table 2 reports the total numbers of lines, words, and characters as reported by the UNIX wc command for each set of these files. It is clear that ASDL description is more compact than the C++ implementation.

A qualitative comparison between the ASDL and alternative systems syntax can be found in Appendix A. It contains an ASDL description and semantically equivalent encoding of the description in various other specification languages (ASN.1, SGML, GMD's ast, and OMG's IDL). Of the specification languages, the ASN.1 specification seems to be comparable in clarity to ASDL. Though syntax is sometimes a matter of taste, the lexical restrictions of ASN.1 makes translation of an ASN.1 identifiers into idiomatic target language identifiers more complicated than necessary.

3.3 ASDL Tools

We have constructed the following tools:

- A prototype definitions generator that reads ASDL descriptions and produces data structures defintions and pickling routines in C, C++, Java, and ML
- A browser-editor that can graphically view and manipulate arbitrary ASDL pickles
- A tool to convert between the original C++ SUIF data structures and data structures produced from the ASDL description by the definitions generator[Ser97].

3.3.1 Prototype Definitions Generator

ASDL has been used to describe the internal data structures of a prototype tool that generates code from ASDL descriptions. Rather than manipulating raw strings, the tool works with data structures that represent the abstract syntax trees (AST) of the target languages (C, C++, Java, ML). The AST is then pretty printed [Opp80] to produce the final output. The tool uses the translation techniques outlined in Section 3. A related tool produces a set of C++ functions that automatically pickle and unpickle the C++ data structures.

During the initial design process of ASDL the first thing discussed was the abstract syntax of ASDL. A proposal for the abstract syntax of ASDL was written using the algebraic data type notation of ML. A purposed concrete syntax was also discussed along with the abstract syntax. The clean separation between the abstract and concrete syntax helped isolate important issues of language design from issues of syntax. Although the initial concrete syntax was substantially modified for the

Original C++ code

class *tree_node*		...
class *tree_block*	:	public *tree_node* $\{ty_1\ \mathrm{f}_1; \ldots; ty_n\ \mathrm{f}_n;\}$
class *tree_proc*	:	public *tree_block* $\{ty'_1\ \mathrm{f}'_1; \ldots; ty'_n\ \mathrm{f}'_n;\}$

Encoding without Attributes

tree_node	=	...
	|	**Tree_block_rest**(*tree_block_rest*)
tree_block_rest	=	**Tree_block**($ty_1\ \mathrm{f}_1, \ldots, ty_n\ \mathrm{f}_n$)
	|	**Tree_proc**($ty_1\ \mathrm{f}_1, \ldots, ty_n\ \mathrm{f}_n, ty'_1\ \mathrm{f}'_1, \ldots, ty'_n\ \mathrm{f}'_n$)

Encoding with Attributes

tree_node	=	...
	|	**Tree_block_rest**(*tree_block_rest*)
tree_block_rest	=	**Tree_block**
	|	**Tree_proc**($ty'_1\ \mathrm{f}'_1, \ldots, ty'_n\ \mathrm{f}'_n$)
		attributes($ty_1\ \mathrm{f}_1, \ldots, ty_n\ \mathrm{f}_n$)

Figure 13: Encoding inheritance in ASDL

class *sym_node*	:	...
class *proc_sym*	:	public *sym_node* { ... *tree_proc* *pr; ... }
sym_node	=	**Proc_sym**(..., **tree_node** pr,...)

Figure 14: Ignoring subtyping constraints

asdl_ty	=	**Sum**(*identifier, field*, constructor, constructor**)
	|	**Product**(*identifier, field, field**)
constructor	=	**Con**(*identifier, field**)
field	=	**Id**(*identifier, identifier?*)
	|	**Option**(*identifier, identifier?*)
	|	**Sequence**(*identifier, identifier?*)

Figure 15: Abstract Syntax of ASDL in ASDL

current version of ASDL syntax, the abstract syntax has changed little from the initial proposal.

The abstract syntax for ASDL can itself be expressed in ASDL (see Figure 15). This is an important property that is used by the browser-editor. Since an ASDL description can be represented as an ASDL value, after a parser converts the concrete syntax of an ASDL description into its abstract form the description can itself be pickled. Other tools can read and manipulate the abstract syntax without any dependence on the concrete syntax. The browser is one such tool.

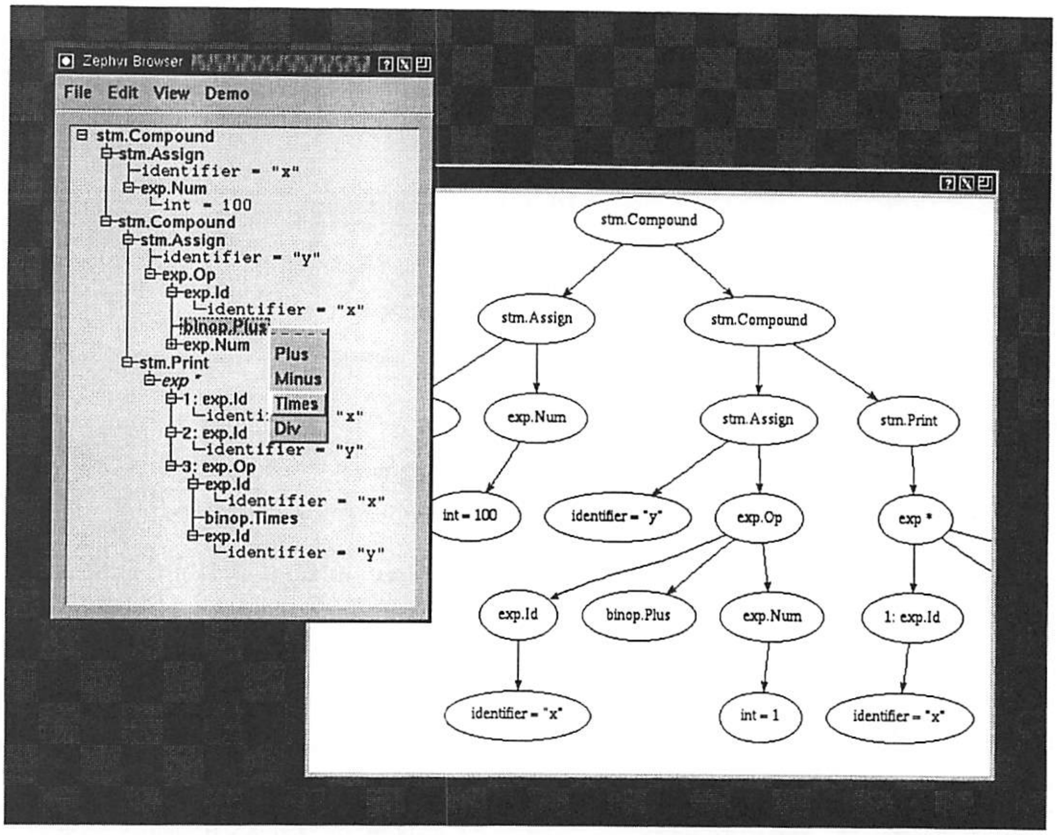
3.3.2 Graphical Pickle Browser and Editor

The browser is a graphical tool for viewing and editing arbitrary pickled ASDL values. The browser reads in two pickles to do this. One is an arbitrary pickled ASDL value. The other it the pickled ASDL description that contains all the ASDL types that occur in the first pickle. Given the ASDL description for the first pickle the brower-editor is able to display the first pickle as a hierarchical list or a graphical tree. It allows the user to specify how each kind of node is drawn by allowing the selection of colors, fonts, etc. Trees can be edited using standard cut and paste operations or by creating/modifying nodes. If the user double-clicks on a particular constructor, a pop-up menu will appear allowing the user to change which constructor-type the node should have. Upon selecting a type for a node, the browser fills in new nodes for the children of that type automatically. Pickles edited with the browser can be saved as a new pickled value.

The browser is written in C. It manipulates a C version of the ASDL abstract syntax produced by the definition generator from the ASDL description of ASDL. When the user edits an object, the browser modifies an abstract representation of generic ASDL values in mem-

	files	lines	words	characters
C++ kernel (with comments)	63	25,095	76,094	632,693
C++ core files (without comments)	10	2,316	6,533	60,984
ASDL description	1	204	562	6,921

Table 2: Code size comparison of SUIF data structures.



```
Zephyr Browser
File  Edit  View  Demo
⊟ stm.Compound
  ⊟-stm.Assign
  │ ├-identifier = "x"
  │ ⊟-exp.Num
  │   └int = 100
  ⊟-stm.Compound
    ⊟-stm.Assign
    │ ├-identifier = "y"
    │ ⊟-exp.Op
    │   ⊟-exp.Id
    │   │ └identifier = "x"
    │   ├binop.Plus
    │   ⊞-exp.Num
    ⊟-stm.Print
      ⊟-exp *
        ⊟-1: exp.Id
        │ └identi:        "x"
        ⊟-2: exp.Id
        │ └identifier = "y"
        ⊟-3: exp.Op
          ⊟-exp.Id
          │ └identifier = "x"
          ├binop.Times
          ⊟-exp.Id
            └identifier = "y"
```

Figure 16: Browser-editor for ASDL pickles

value	=	**SumVal**(*identifier*, *value**, *value**)
	|	**ProductVal**(*value*, *value**)
	|	**SequenceVal**(*value**)
	|	**NoneVal**
	|	**SomeVal**(*value*)
	|	**PrimVal**(*priml*)
prim	=	**IntVal**(*int*)
	|	**IdentifierVal**(*identifier*)
	|	**StringVal**(*string*)

Figure 17: ASDL values specified in ASDL

ory, also generated from the ASDL description in Figure 17. The in memory representation is then converted into the correct pickle format. This makes the browser completely independent of the actual details of both the concrete syntax of ASDL or the actual pickling format. As long as the abstract structure of the pickles and concrete syntax stay unchanged, changes to the details of the pickle format or concrete syntax have little impact on the browser. Since ASDL descriptions are also pickled values the browser can be used to create, edit, or view ASDL type descriptions by manipulating the abstract syntax of ASDL directly. The browser-editor and parser for the concrete syntax of ASDL are two different user interfaces to building the abstract syntax of ASDL descriptions.

Since the ASDL approach allows applications written in different programming languages to interoperate, the browser is implemented in C allowing the use of an existing and advanced GUI toolkit such as Tcl/Tk. The definitions generator and the parser for ASDL descriptions are implemented in Standard ML, but these two tools easily interoperate with each other because of the standard pickle format. ASDL gives developers more flexibility in choosing the appropriate language for a given task.

4 Related Work

Automatically encapsulating runtime data structures into external out of core values is not an idea unique to ASDL. Some languages, notably Modula-3, have built-in language support for translating values into "pickles". Unlike ASDL, Modula-3 pickling is able to handle arbitrary graph structures and does pickling based on runtime type information and support from a garbage collector. ASDL pickling is based on compile time static information, which makes it less flexible than Modula-3 but more portable and efficent. ASDL pickling code need not depend on any special runtime support and can be optimized based on static information. Java has borrowed the pickling techniques and ideas from Modula-3 to provide the automatic "serializing" of arbitrary language objects.

Pizza [OW97] is a superset of Java that provides algebraic data types, giving Java concise notation for tree-like data structures. A combination of Pizza along with the automatic serialization of types in Java has some similarities to the ASDL approach. Unfortunately the pickles that Java produces are not meant to be language independent.

The most similar work to ASDL is ASN.1. Ignoring syntactic issues, ASDL, resembles a subset of ASN.1. The original evaluation of the existing systems suggests that ASN.1 can solve the component interoperation problem. Commercial tools exist that translate ASN.1 into C, C++, and Java, so it is tempting to use ASN.1 and write the remaining tools for languages not supported by commercial systems. However, ASDL has one significant advantage over ASN.1. ASDL is much simpler than the full ASN.1 specification language. Simplicity is not just an esthetic concern. The complexity of ASN.1 makes it difficult to write tools that use it. Considering ASDL is able to solve the problems in the compiler domain, the extra effort in dealing with the complexity of ASN.1 complexity does not seem worth the effort.

The official context free grammar of the most recent version of ASN.1 contains over 150 non-terminals and 300 productions, and occupies eight pages [ITU95b]. The equivalent ASDL context free grammar contains a little over ten non-terminals, twenty productions, and easily fits on half a page. The size of the ASN.1 grammar alone makes it difficult to build tools for it. A freely available ASN.1 compiler [Sam93], which converts a subset of ASN.1 to C and C++ but parses the full ASN.1 language, has a 3000 line yacc grammar. The rest of the system consists of 13000 lines of C. Because of ASDL's simplicity, it is easy to construct a definitions generator for different languages. The prototype tool described in this document took a few weeks to implement and is around 5000 lines of ML code. This prototype generates definitions for four different languages.

Although ASN.1 is intended to describe network data, it is also used to describe data in other domains, such as chemical abstracts [CXF94] and gene sequences [NCB96]. These ASN.1 specifications exist to help improve the exchange of information across software systems written to manipulate data in these domains. Close inspection of the ASN.1 specifications for these domains reveals that they only use a small subset of the features of ASN.1 and that the subset they use is very close to ASDL. This suggests that ASDL has wider applications, and that it is worthwhile to develop a strategy to interoperate with existing systems using ASN.1.

5 Future Work

As the SUIF encoding demonstrates, realistic ASDL descriptions may still be reasonably long. ASDL should support modularized descriptions. Modularizing descriptions at the ASDL level requires us to address the issue of how modular descriptions are translated into the target language. Should each module of the description correspond to a compilation unit in the target language? Should cyclic module dependencies be allowed? Cyclic module dependencies are convenient when describing ASTs, but languages like ML do not support cyclic module dependencies.

More work needs to be put into building tools that use the ASDL definitions. Our current prototype tool performs a naive translation of an ASDL description into target languages. The tools that generate ASDL descriptions need to have more hooks so that users can control how descriptions are translated. Tools could also perform more aggressive automatic representation optimizations on the generated code.

It seems appropriate to reuse ASDL descriptions for a wide variety of other tools, such as attribute evaluators, parsers, pattern matchers, and pretty printers. All these systems can benefit from the formalisms that ASDL provides. Jansson [JJ97] presents a formalism (polytypic programming) to describe functions that generate functions based on structural induction on an arbitary algebraic data type. Polytypic programming allows the creation of generator generators. Jansson's approach could be extended into a tool that generates code generators for ASDL data types. A polytypic description that inductively describes the equality of arbitary types can be turned automatically into a program that takes an ASDL description and produces another program to check for equality.

6 Conclusions

Declarative languages such as regular expressions and context-free grammars, with tools like `lex` and `yacc`, help popularized the notion of describing the concrete syntax of programming languages formally. Description languages like ASDL will popularize the notion of formally describing the abstract syntax of programming languages and the internal representations of compilers. Our initial experience with SUIF and other descriptions suggests that ASDL is able to encapsulate the fundamental structures of important data structures in a concise and language independent way.

The core idea of concise notation for describing tree like-data structures behind ASDL is so simple it has been reinvented in different guises by several systems that span a variety of domains. ASN.1 is one such system, which also provides support for cross language interoperation. Unfortunately the full ASN.1 language is complex, making tool development a difficult task. ASDL represents a simple and powerful subset of ASN.1. The simplicity of ASDL allows for easier implementation of tools that use it.

7 Availability

ASDL is part of a larger project to develop a resuable compiler infrastructure. The most up to date information on ASDL can be found at `http://www.cs.virginia.edu/zephyr`. We are currently working on a production quality release of the software and documentation, which should be completed in January 1998. Working releases of the software will be available in the interim; see the web page for details.

8 Acknowledgments

We would like to thank all those who have helped out along the way including the developers of SUIF, Norman Ramsey, and David Hanson.

9 Glossary of Acronyms

ASDL	Abstract Syntax Description Language
ASN.1	Abstract Syntax Notation One
AST	Abstract Syntax Tree
CORBA	Common Object Request Broker Architecture
EBNF	Extended Backus Naur Form
IDL	Interface Description Language
ILU	Inter Language Union
IR	Intermediate Representation
ISL	Interface Specification Language
ONC	Open Network Consortium
OMG	Object Management Group
PER	Packed Encoding Rules
RPC	Remote Procedure Call
SGML	Standard Generalized Markup Language
SUIF	Stanford University Intermediate Format
XDR	External Data Representation

References

[Bat96] Rodney M. Bates. Examining the Cocktail toolbox. *Dr. Dobb's Journal of Software Tools*, 21(3):78, 80–82, 95–96, March 1996.

[BMS80] R. Burstall, D. MacQueen, and D. Sannella. Hope: an experimental applicative language. In *Proceedings of the 1980 LISP Conference*, pages 136–43, Stanford, 1980.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 217–230, 1993.

[CXF94] Chemical exchagne format. `ftp://info.cas.org/pub/cxf`, 1994.

[FH95] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[GHL+92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, February 1992.

[GR90] Charles F. Goldfarb and Yuri Rubinsky. *The SGML handbook*. Clarendon Press, Oxford, UK, 1990.

[ISO87] Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1). International Organization for Standardization and International Electrotechnical Committee, 1987. International Standard 8824.

[ITU95a] Information Technology – Abstract Syntax Notation One (ASN.1): Encoding Rules – Packed Encoding Rules (PER). International Telecommuncation Union, 1995. ITU-T Recommendation X.691.

[ITU95b] Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. International Telecommuncation Union, 1995. ITU-T Recommendation X.680.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 15–17 January 1997.

[JPJ+90] M. Jourdan, D. Parigot, C. Julie, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 209–222, White Plains, New York, June 1990.

[NCB96] National center for biotechnology software development toolkit. `ftp://ncbi.nlm.nih.gov/toolbox/ncbi_tools`, 1996.

[Obj95] Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, 1995.

[Opp80] Dereck C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings POPL 1997*, Paris, January 15-17 1997.

[Sam93] Michael Sample. Snacc 1.1. `http://www.nsg.bc.ca/Software.html`, 1993.

[Ser97] Christopher S. Serra. Bridging suif and zephyr:a compiler infrastructure interchange. Princeton University Senior Thesis, May 1997.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[Sri95] Raj Srinivasan. RFC 1831: RPC: Remote Procedure Call Protocol specification version 2, August 1995.

[Vol91] J. Vollmer. Experiences with gentle: Efficient compiler construction based on logic programming. *Lecture Notes in Computer Science*, 528:425–??, 1991.

[W+94] Robert Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[Wir77] N. Wirth. What can be do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):882, November 1977.

[Xer96] Xerox Corporation. *ILU 2.0alpha8 Refernce Manual*, May 1996. `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[XML97] Extensible markup language (XML). `http://www.w3.org/TR/WD-xml`, 1997.

A Appendix A

A.1 Zephyr ASDL

```
stm = Compound(stm head, stm next)
    | Assign(identifier id, exp exp)
    | Print(exp* args)
exp = Id(identifier id)
    | Num(int v)
    | Op(exp lval, binop bop, exp rval)
binop = Plus | Minus | Times | Div
```

A.2 GMD's compiler toolkit

```
-- See http://www.gmd.de/SCAI/lab/adaptor/ast.html
Stm = <
   Compound = head: Stm next: Stm .
   Assign   = [id: char*] exp: Exp .
   Print    = args: ExpList .
> .
Exp = <
   Id  = [id: char*] .
   Num = [v: int] .
   Op  = lval: Exp bop: Binop rval: Exp .
> .
Binop = < Plus = .
          Minus = .
          Times = .
          Div   = .
        > .
```

A.3 ISO X.680 ASN.1

```
-- See http://www.itu.ch/itudoc/itu-t/rec/x/x500up/x680_27252.html
Stm ::= CHOICE {
         compound SEQUENCE {head Stm, next Stm},
         assign SEQUENCE {head Stm, next Stm},
         print SEQUENCE {args SEQUENCE OF Exp}
        }
Exp ::= CHOICE {
         id STRING,
         num INTEGER,
         op SEQUENCE {lval Exp,bop BinOp,rval Exp}
        }
Binop ::= ENUMERATED {plus, minus, times, div}
```

A.4 SGML DTD

```
<!ENTITY % id  "(#PCDATA)">
<!ENTITY % int  "(#PCDATA)">
<!ENTITY % binop "(Plus|Minus|Times|Div)">
<!ENTITY % stm  "(Compound|Assign|Print)">

<!ELEMENT Compound - - (%stm,%stm)>
<!ELEMENT Assign - - (%id,%exp)>
<!ELEMENT Print - - (%exp*)>

<!ENTITY % exp  "(Id|Num|Op)">
<!ELEMENT Id - - (%id)>
<!ELEMENT Num - - (%int)>
<!ELEMENT Op - - (%exp,%binop,%exp)>
```

A.5 OMG IDL Object Encoding

```
-- http://www.omg.org/corba/corbiiop.htm
enum binop { Plus, Minus, Times, Div};
interface stm {
        enum stm_tag { Compound_tag, Assign_tag, Print_tag};
        attribute stm_tag tag;
}
interface Compound : stm {
        attribute stm head; attribute stm next;
}
interface Assign : stm {
        attribute id string; attribute exp exp;
}
interface Print : stm {
        attribute sequence<exp> args;
}
interface exp {
        enum exp_tag { Id_tag, Num_tag, Op_tag};
        attribute exp_tag tag;
}
interface Id  : exp { attribute id string; }
interface Num : exp { attribute int v; }
interface Op  : exp {
        attribute exp lval; attribute binop bop; attribute exp rval;
}
```

ASTLOG: A Language for Examining Abstract Syntax Trees

Roger F. Crew

Microsoft Research
Microsoft Corporation
Redmond, WA 98052
rfc@microsoft.com

Abstract

We desired a facility for locating/analyzing syntactic artifacts in abstract syntax trees of C/C++ programs, similar to the facility `grep` or `awk` provides for locating artifacts at the lexical level. Prolog, with its implicit pattern-matching and backtracking capabilities, is a natural choice for such an application. We have developed a Prolog variant that avoids the overhead of translating the source syntactic structures into the form of a Prolog database; this is crucial to obtaining acceptable performance on large programs. An interpreter for this language has been implemented and used to find various kinds of syntactic bugs and other questionable constructs in real programs like Microsoft SQL server (450Klines) and Microsoft Word (2Mlines) in time comparable to the runtime of the actual compiler.

The model in which terms are matched against an implicit current object, rather than simply proven against a database of facts, leads to a distinct "inside-out functional" programming style that is quite unlike typical Prolog, but one that is, in fact, well-suited to the examination of trees. Also, various second-order Prolog set-predicates may be implemented via manipulation of the current object, thus retaining an important feature without entailing that the database be dynamically extensible as the usual implementation does.

1 Introduction

Tools like `grep` and `awk` are useful for finding and analyzing lexical artifacts; e. g., a one-line command locates all occurences of a particular string. Unfortunately, many simple facts about programs are less accessible at the character/token level, such as the locations of assignments to a particular C++ class member. In general, reliably extracting such syntactic constructs requires writing a parser or some fragment thereof. And after writing one's twenty-seventh parser fragment, one might begin to yearn for a more general tool capable of operating at the syntax-tree level.

Even given a compiler front-end that exposes the abstract syntax tree (AST) representation for a given program, there remains the question of what exactly to do with it. To be sure, supplying a C programmer with a sufficiently complete interface to this representation generally solves any problem one might care to pose about it. One may just as easily say that all problems at the lexical level may be solved via proper use of the UNIX standard IO library `<stdio.h>`, a true, but utterly trivial and unsatisfying statement. The question is rather that of building a simpler, more useful and flexible interface: one that is less error-prone, more concise than writing in C, and more directly suited to the task of exploring ASTs. We first consider a couple of prior approaches.

1.1 The `awk` Approach

One of the more popular approaches is to extend the `awk` [AKW86] paradigm. An `awk` script is a list of pairs, each being a regular-expression with an accompanying statement in a C-like imperative language. For each line in the input file, we consider each pair of the script in turn; if the regular-expression matches the line, the corresponding statement is executed.

Extending this to the AST domain is straightforward, though with numerous variations. One defines a regular-expression-like language in which to express tree patterns and an `awk`-like imperative language for statements. The tree nodes of the input program are traversed in some order (e.g., preorder), and for each node the various pairs of the script are considered as before.

We have two objections to this approach, the first having to do with the hardwired framework that usually implicit. In some cases (e. g., TAWK [GA96]), the traversal order for the AST nodes is essentially fixed; using a different order would be analogous to attempting to use plain `awk` to scan the lines of a text file in reverse order. In A* [LR95], while the user may define a general traversal order, only one traversal method may be defined/active at any given time, making difficult any structure comparisons between subtrees or other applications that require multiple concurrent traversals. Since the imperative language is quite general in both cases, little is definitively impossible, however for some applications one may be little better off than when programming in straight C.

The second objection has to do with the kinds of pattern-abstraction available. Inevitably there exist simply-described patterns that are a poor fit to a regular-expression-like syntax. This tends to happen when said simple descriptions are in terms of the idioms of a particular programming language; most of the various tree-`awk` pattern languages tend to be designed with the intent of being language independent.

Suppose one wishes to find all consecutive occurrences of one statement immediately preceding another, e. g., places where a given system call `syscall();` is followed immediately by an `assert();` (on the theory that testing of outcomes of system calls should be done in production code rather than just debugging code). A tree-regular-expression pattern of the form

$$\langle\texttt{syscall()}\ \text{pattern}\rangle;\langle\texttt{assert()}\ \text{pattern}\rangle$$

(where ; is the regular-expression sequence operator) finds all instances of the two calls occurring consecutively within a single block, but it misses instances like

```
syscall();
{
  assert();
  ...
}
```

and

```
if (...) {
  syscall();
}
else {
  ...
}
assert();
```

While the tree-`awk` languages allow one to write patterns to match each of these cases, without a pattern-abstraction facility, we may be back at square one when it comes time to look for some *different* pair of consecutive function calls. We prefer to write a single consecutive-statement pattern-constructor *once* and then be able to use it for a variety of cases where we need to find pairs of consecutive statements satisfying certain criteria, invoking it as

$$\texttt{follow\_stmt}(\langle\texttt{syscall()}\ \text{pattern}\rangle,\ \langle\texttt{assert()}\ \text{pattern}\rangle)$$

for the above problem, or, if we instead want to be finding all of the places where a C switch-case falls through, as

$$\texttt{follow\_stmt}(\texttt{not}(\langle\text{unconditional-jump pattern}\rangle),\ \langle\text{case-labeled stmt pattern}\rangle)$$

One solution, used by TAWK, is to use `cpp`, the C preprocessor, to preprocess the script, allowing for pattern-abstractions to be expressed as `#define` macros whose invocations are then expanded as needed. This is unsatisfactory in a number of ways, whether one wants to consider the problem of recursively-defined patterns, macros with large bodies that result in a corresponding blow-up in the size of the script, or the difficulty of tracing script errors that resulted from complex macro-expansions.

Another way out is to fall back on the procedural abstraction available in the imperative language that the patterns invoke. One essentially uses a degenerate pattern that always matches and then allows the imperative code to test whether the given node is in fact the desired match, defining functions to test for particular patterns. Once again, it seems we are back to programming in straight C and not deriving as much benefit from having a pattern language available as we could be.

In general, the philosophical underpinning of the `awk` approach is that the designer has already determined the kinds of searches the user will want to do; the effort is put towards making those particular searches run efficiently. There is also an assumption that the underlying imperative language for the actions has all the abstraction facilities one will ever need, so that if the pattern language is lacking in various ways, this is not deemed a serious problem. While this is not an unreasonable approach, we have less confidence of having identified all of the reasonable search possibilities, and thus would prefer instead to make the pattern language more flexible and extensible, being willing to sacrifice some efficiency to do so.

1.2 The Logic Programming Approach

Another common approach is to run an inference engine over a database of program syntactic structures [BCD+88, BGV90, CMR92]. Prolog [SS86] is a convenient language for this sort of application. Backtracking and a form of pattern matching are built in, the abstraction mechanisms to build up complex predicates exist at a fundamental level, and finally, Prolog allows for a more declarative programming style.

script	::= named-clause*	script file syntax
query	::= imports? (varname*) clause-body ;	query syntax
imports	::= { varname+ }	
named-clause	::= opname anon-clause	
anon-clause	::= (term*) clause-body? ;	
clause-body	::= <- term+	
	Essential Term Syntax	
term	::= literal	reference to denotable object
	::= varname	
	::= opname (term*)	compound term
	::= `FN` imports? (anon-clause+)	anonymous predicate-operator-valued ("lambda") term
	::= ' opname arity-spec?	named predicate-operator-valued ("function quote") term
	::= (term) (term*)	"application" term
	Gratuitous Term Syntax	
	::= # constname	named constant (≡ corresponding literal number)
	::= [term*]	`[ ]` ≡ `nil()`, `[`term`]` ≡ `cons(`term`,nil())`, etc...
	::= [term+ | term]	`[` term1 `|` term2 `]` ≡ `cons(`term1`,`term2`)`, etc...
arity-spec	::= / integer	

Figure 1: Complete Syntax of ASTLOG

The problems with using Prolog are two-fold. First there is the issue of efficiency. Second, we must represent the AST for our source program in the Prolog database. Large programs ($10^5 - 10^6$ lines) will result in correspondingly large Prolog databases, most likely with a significant performance penalty.

We finesse the second problem by not attempting to import the source program's AST at all, instead opting to modify the interpretation of the predicates and queries of Prolog so as to be applicable to external objects rather than just facts provable in the existing database. Removing reasons that require the database to grow beyond the initial script creates significant opportunities for optimization. This, however, requires removing primitives like `assert()` and `retract()` that allow for the dynamic (re)definition or removal of predicates, which in turn removes many higher-order logical features that are defined in terms of them. Fortunately, some of the more essential ones can be restored at relatively little cost.

2 Elements of ASTLOG

Figure 1 gives the complete syntax for our language, ASTLOG. The ASTLOG interpreter reads a script of user-defined predicate operator definitions and then runs one or more queries.

As in Prolog, the definition of a user-defined predicate operator is composed of one or more *clauses*. A compound term *opname*(*term*, ...) appearing at top level in a clause body is interpreted as a predicate, whether *opname* be primitive or user-defined. In the latter case, the script is searched for a defining clause whose head terms successfully unify with the respective operand terms of the given compound term, variables are bound accordingly, and the terms of the clause body are likewise interpreted. The clause *succeeds* (i. e., is found to be true) if all of its body terms succeed. Whenever a clause head fails to unify, or a clause body term is *fails* (i. e., is found to be false), or any primitive term fails by the rules of evaluation of that primitive, we backtrack to the last point where there was a choice (e. g., of clauses to try for a given compound term) and continue.

A *query* is a clause whose head terms are all variables. Ultimately, whenever all terms of a query body succeed, the bindings of any variables listed in the query head (*qhead*) are reported. Otherwise, we report failure. Thus far, this is all exactly like Prolog.

2.1 Objects

ASTLOG refers to external objects. Given a C/C++ compiler front end that provides a (C++) interface to the syntactic/semantic data structures built during the parse of a given program, it is simple to graft this onto the core of ASTLOG so that it may recognize object references corresponding to

- whole C/C++ programs,
- single files,
- symbols,
- AST nodes (including statements, expressions, and declarations), and

- C/C++ type descriptions.

For the purposes of ASTLOG, an *object* is simply some external entity that is significant for its identity and for the primitive predicates that it may satisfy. To simplify the language we regard the traditional constants (integers, floats, and strings) to be references to "external" objects as well, though one could just as easily take the converse view in which the universe of object references is just a (very large) pool of constants ("atoms" in the usual Prolog terminology).

In any case, object references are terms in ASTLOG. Only references to equal objects can unify, equality meaning numeric equality for numbers, same-sequence-of-characters for strings, and identity for all other classes of objects. Only objects that have denotations (numbers, strings and the unique null object `*`) can find their way into scripts.

2.2 The Current Object

The first significant departure from the Prolog model is that a query or predicate term always evaluates under an ambient *current object*. Every query and every term being evaluated as a predicate is not so much a stand-alone statement that may or may not be intrinsically true (i. e., provable from the "facts" in the script) as it is a specification that may or may not be satisfied by the current object, or, alternatively, a *pattern* that may or may not *match* the current object. For example, in Prolog

```
odd(3)
```

always succeeds by virtue of 3 being odd or because the "fact" `odd(3)` exists in the script somewhere. By contrast, in ASTLOG

```
odd()
```

succeeds if the current object happens to be the integer 3, fails if the current object is 4, and raises an error if the current object is the string `"Hi mom"`. Another way to view this is that every predicate term takes an extra, hidden current-object operand.

While one normally only expects to see compound (and application) terms in predicate position, ASTLOG allows variables and object references there as well. The rules for matching are as follows:

- An object reference matches the current object iff it references an equal object.
- A bound variable matches according as whatever term it is bound to.
- An unbound variable gets bound to reference the current object (and thus automatically matches it).
- A compound term whose operator is defined via clauses matches iff there exists a clause whose head operands unify with the term operands and whose body terms themselves all match the current object.

Section 3.1 describes the operator-valued and application terms.

The evaluation rules for compound terms having primitive operators are widely varied, however the operands are usually treated one of two ways:

1. (*foo-pred*) requiring the operand to be match some object (which becomes the current object for that evaluation), not necessarily the same current object as that which the full term is being matched against. For example, the operand of `strlen` (see Figure 2) and the second operand of `with` are treated this way.
2. (*foo*) requiring the operand be an object reference, whether this be a literal or an object-reference-bound variable. The operands of `re`, `gt`, and the first operand of `with` are treated this way.

Most primitives also expect a current object to be of a particular kind and raise an error if confronted with something different.

The use of an implicit current object is not by itself an increase in expressivity. If we had, in a Prolog database, terms representing the various AST nodes, there would be a fairly straightforward translation of ASTLOG terms into Prolog terms, one in which we simply modify all terms to make the current object an explicit operand.

Nevertheless, ASTLOG programs exhibit a distinct style of programming. Consider as an example that we might, in a typical functional language, write a function call

strlen(*string*)

to find the length of the string returned by the expression *string*. Here the length result is implicitly returned to the context of the call. In Prolog, the natural style would be to express this as a relation

```
strlen(string, length)
```

which stipulates that `length` is in fact the length of `string`. In ASTLOG, we would write

strlen(*length-pred*)

where now it is the string argument that is implicitly supplied (as the current object) *by* the context while the length result is returned *to* the subterm *length-pred*, which in turn can be some arbitrary term expecting a numeric current object as its implicit argument. For example, given an `odd()` predicate as

- `and`(*object-pred*, ...)
  The current object satisfies every *object-pred* operand.
- `or`(*object-pred*, ...)
  The current object satisfies some *object-pred* operand.
- `if`(*object-pred*, *then-pred*, *else-pred*)
  The current object satisfies *then-pred* or *else-pred* according as it satisfies or fails to satisfy *object-pred* (once; if *object-pred* matches but *then-pred* does not, we do not retry *object-pred*).
- `not`(*object-pred*)
  = `if`(*object-pred*, `or`(), `and`())
- `with`(*object*, *object-pred*)
  object satisfies *object-pred* (outer current object is ignored).
- `strlen`(*integer-pred*)
  The current string object has length satisfying *integer-pred*.
- `re`(*string*)
  The regular expression *string* matches the current string.
- `gt`(*integer*)
  The current integer is greater than *integer*.
- `minus`(*integer-pred*, *integer*)
  integer-pred matches the current integer + *integer*.
- `minus`(*integer*, *integer-pred*)
  integer-pred matches *integer* − the current integer. (An error is raised if neither operand of a `minus` term is an integer object reference.)
- `plus`(*integer-pred*, *integer*)
  integer-pred matches the current integer − *integer*.

Figure 2: Some core ASTLOG primitives

above, the term `strlen(odd())` would match any string consisting of an odd number of characters. It is this "inside-out functional" evaluation strategy that makes ASTLOG well-suited to constructing anchored patterns to match tree-like structures.

2.3 Examples

Given the set of AST node primitives in Figure 3, we could write

```
and(op(#=), kid(#LEFT, asym(sname("foo"))))
```

which would be satisified by any AST node that is an assignment (`=`) expression whose left-hand side is itself a symbol expression where the symbol name is `"foo"`. Here, `#=` and `#LEFT` are numeric literals for the assignment node opcode and the assignment target's child-index, respectively.

To define a predicate `assignment/2` to match assignment nodes, a script could include the clause

```
assignment(target, value)
        <-  op(#=),
            kid(#LEFT, target),
            kid(#RIGHT, value);
```

which would then allow writing the previous term as

```
assignment(asym(sname("foo")), _)
```

As in Prolog, the underscore (`_`) is "wild-card" variable, i. e., one that is internally given a distinct identity so as not to be conflated with any other instances of `_`. Such a variable, being guaranteed to be unbound, will match any object or unify with any term.

Defining a general purpose node-traversal predicate is also straightforward

```
somenode(pred)
        <-  or(pred, kid(_, somenode(pred)));
```

Given this definition, an attempt to match `somenode(`*test*`)` to a given node will create an instance of the defining clause of `somenode/1` above with `pred` bound to *test*. Satisfying the clause body requires that either `pred` match the current node, or, if (when) that fails, that `kid(_,somenode(pred))` match the current node. The latter in turn will attempt to match the variable `_` with 0 (easy) and the term `somenode(pred)` with the first child, and, when that fails, `_` with 1 and `somenode(pred)` with the second child, etc... Making the interpreter fail and backtrack after each hit (in the usual manner of Prolog) eventually causes *test* to be matched with the original node and all of its descendants.

- `parent`(*ast-pred*)
  This AST node is not a root node and its parent satisfies *ast-pred*.
- `kid`(*integer-pred*, *ast-pred*)
  This AST node has a child satisfying *ast-pred* whose (0-based) index satisfies *integer-pred*.
- `kidcount`(*integer-pred*)
  The number of children of this AST node satisfies *integer-pred*.
- `op`(*integer-pred*)
  The opcode of this AST node satisfies *integer-pred*.
- `atype`(*type-pred*)
  This AST node has a return type satisfying *type-pred*.
- `asym`(*symbol-pred*)
  This AST node is a symbol satisfying *symbol-pred*.
- `aconst`(*const-pred*)
  This AST node is a constant (integer, float or string) satisfying *const-pred*.
- `sname`(*string-pred*)
  This symbol's name satisfies *string-pred*.

There are named constants available for designating the opcodes of various kinds of nodes for use in `op()` terms, and the indices of particular children for use in `kid()` terms.

Figure 3: Some primitive node and symbol predicates

So, if we issue the query

```
   (v) <- somenode(
        assignment(asym(sname("foo")), v)
                        );
```

on the root node of some function's AST, we obtain, via the successive bindings reported for `v` on each hit, all of the expressions assigned to variables named `"foo"` within that function.

As an example that makes less trivial use of backtracking, consider the problem of whether two trees have the same structure (i. e.., root nodes have the same opcode and all corresponding children have the same structure).

```
sametree(node)
      <-  op(nodeop),
          with(node, op(nodeop)),
          not(and(with(node, kid(n, nkid)),
                  kid(n, not(sametree(nkid)))));
```

This defines a predicate `sametree`(*node*) that holds iff *node* is a reference to an AST node with the same structure as the current object. The first line of the clause body binds the current node's opcode to `nodeop`, the second line compares that to the opcode of `node`, while the remaining lines search for children whose subtrees have distinct structure. The term `kid(n,nkid)` will match each child of `node`, since both variables are initially unbound. If `sametree(nkid)` happens to be true of the corresponding child of the current node, the inner `not` fails and we go back and try another child of `node`. If `sametree(nkid)` happens to be true of *every* corresponding child of the current node, then the enclosing `not` and thus the outer `sametree(node)` invocation succeeds.

The preceding version of `sametree/1` is a purely structural comparison; there is no attempt to take account of the commutativity/associativity of the various operators, e. g., a + b and b + a are not considered the same. If, say, we *did* want to consider commutativity, we could define

```
 csametree(node)
   <- op(nodeop),
      with(node,op(nodeop)),
      kidcount(if(with(nodeop,commutes()),
                  any_perm(perm),
                  id_perm(perm))),
      not(and(with(node,kid(corresp(perm,n),
                                nkid)),
              kid(n,not(csametree(nkid)))));
```

along with suitable definitions of

`commutes()`
: the current integer is the opcode of a commutative operator,

`any_perm`(*perm*)
: *perm* is any permutation of the sequence 0, ..., (⟨current-object⟩ − 1),

`id_perm`(*perm*)
: *perm* is the identity permutation of the sequence 0, ..., (⟨current-object⟩ − 1),

`corresp`(*perm*, *n*)
: permutation *perm* takes the current integer to something matching *n*.

Here, permutations can be represented by list terms. Note that since all of the commutative C/C++ operators are, in fact, binary, this all simplifies significantly.

It should, incidentally, be clear that there is nothing about the core language that is specifically tailored for the examination of compiler-produced ASTs, let alone ASTs for a given language. The language in fact lends itself to the examination of a wide variety of external structures, e. g., hierarchical file systems, or collections of web pages. All that is needed is a suitable collection of primitive ASTLOG predicates for querying said structures.

```
// FOLLOW_STMT(P1 P2)
//   <=> P1 and P2 are true of consecutive statements in this AST

follow_stmt(p1, p2)
  <- if(op(#FUNCTION),
        kid(#FUNCTION/BODY, follow_stmt(p1,p2,*)),
        follow_stmt(p1,p2,*));

follow_stmt(p1, p2, after)
  <- cond(op(#BLOCK),     follow_block_stmt(p1, p2, after),

          op(#IF),        kid(not(#IF/PRED),follow_stmt(p1, p2, after)),
          op(#SWITCH),    kid(#SWITCH/BODY, follow_stmt(p1, p2, after)),

          op(#WHILE),     follow_iter_stmt(#WHILE/BODY,p1, p2, after),
          op(#DO),        follow_iter_stmt(#DO/BODY,    p1, p2, after),
          op(#FOR),       follow_iter_stmt(#FOR/BODY,   p1, p2, after),

          or(op(#LABEL),op(#CASE),op(#DEFAULT)),
           kid(#LABELSTMT/STMT, follow_stmt(p1, p2, after)),

          follow_simple_stmt(p1, p2, after));

follow_simple_stmt(p1, p2, after)
  <- with(after, not(*)), p1, with(after, first_stmt(p2));

follow_iter_stmt(nbody,p1,p2,after)
  <- or(follow_simple_stmt(p1, p2, after),
        and(this, kid(nbody, follow_stmt(p1, p2, this))));

follow_block_stmt(p1, p2, after)
  <- and(kid(minus(next,1), first),
         if(kid(next, second),
            with(first, follow_stmt(p1, p2, second)),
            with(first, follow_stmt(p1, p2, after))));

first_stmt(p)
  <- if(op(#BLOCK),
        kid(0,first_stmt(p)),
        stmt);

// CASEFALL()
//   emits all locations of switch-case fallthroughs in this AST tree
casefall()
  <- follow_stmt(and(not(op(or(#BREAK,#CONTINUE,#GOTO,#RETURN))),first),
                 op(#CASE)),
     with(first,sfa(emit("Fall through to next case.")));
```

Figure 4: Actual ASTLOG code for `follow_stmt` and how one uses it to find case statement fallthroughs. The `cond` operator is an if-then-elseif- construct, that is, `cond`($p_1, e_1, p_2, e_2, \ldots, e$) is equivalent to `if`($p_1, e_1,$ `if`($p_2, e_2, \ldots e$)). `sfa(emit(`*string*`))` always succeeds and, as a side-effect, emits the source location of the current AST node in grep-output form.

```
flatten(test, lst)
  <- flatten(test, lst, []);

flatten(test, head, tail)
  <- if(test,
        first(head, hrest),
        unify(head, hrest)),
     flattenkids(test, 0, hrest, tail);

flattenkids(test, n, head, tail)
  <- if(kid(n, flatten(test, head, mid)),
        and(with(n, minus(nplus1,1)),
            flattenkids(test, nplus1,
                        mid, tail)),
        unify(head, tail));

first([o|rest],rest) <- o;
unify(x,x);
```

Figure 5: Definition of `flatten`

3 Higher order features

We have already included some of the non-1st-order features of Prolog, notably "cut" (in the form of `if()`) and the corresponding notion of negation, `not()`. There are others that turn out to be essential as well.

3.1 Lambdas and Applications

One may observe that, in `somenode`(*test*), because this is an existential query, it does not matter that we are matching the same term *test* to every node of the tree. If variables in *test* get bound as a result of matching a given node, those bindings will be undone prior to advancing to the next node.

If one instead wants to write a conjunctive predicate over all tree nodes, say

`flatten`(*test*, `list`)

which holds if `list` is a list of *all* descendant nodes satisfying *test*, — we give a definition in Figure 5 — this will not work correctly if *test* contains any variables that are bound during the course of matching any node; said variables will *stay* bound for the duration of the `flatten` evaluation.

Even in an existential query, there is the possibility that the *test* being passed in will itself need to take a parameter. For example, one might imagine defining a version of `sametree` that also requires an additional user-specified *test* to hold at each corresponding pair of nodes. If *test* is a mere compound term, it can be matched against one of the nodes, but not both.

Thus we introduce "application" terms and operator-valued terms ("lambdas"). For an application (*fterm*)(*term*, ...) to match the current object, the term *fterm* must be (or be a variable bound to) a predicate-operator-valued term, which will either be

- a reference, `'foo/3` to a named predicate operator, in which case the application evaluates exactly as the corresponding compound term would, or
- an anonymous predicate operator `FN{`*importvars...*`}(`*anon-clauses...*`)`, in which case the application evaluates *almost* exactly as if there were a named predicate-operator defined by the given clauses and this were a compound term on that operator. The difference is that any variables of those clauses that are also on the `{`*importvars...*`}` list are identified with the correspondingly-named variables in the clause where the `FN` term occurs lexically.

An `FN` term with imports can be thought of as a kind of *closure*.

The parameterized version of `flatten`, namely

`flatten2`(*test*, `list`)

which holds iff `list` is a list of all x corresponding to descendants that (*test*)(x) matches, is defined in Figure 6.

```
flatten2(test, lst)
  <- flatten2(test, lst, []);

flatten2(test, head, tail)
  <- if((test)(value),
        unify(head, [value|hrest]),
        unify(head, hrest)),
     flatten2kids(test, 0, hrest, tail);

flatten2kids(test, n, head, tail)
  <- if(kid(n, flatten2(test, head, mid)),
        and(with(n, minus(nplus1,1)),
            flatten2kids(test, nplus1,
                         mid, tail)),
        unify(head, tail));

unify(x,x);
```

Figure 6: Parameterized version, `flatten2`

The parameterized version of `sametree` is invoked as

`sametree`(*node*, *equiv*)

which holds iff *node* is a reference to an AST node with the same tree structure as the current node *and*, for every descendant n of *node*, the corresponding node in the current tree satisfies *equiv*(n); this predicate is defined in Figure 7. This definition demonstrates the use

```
sametree(node,equiv)
 <- unify(same,
     FN{same,equiv}
       ((node)
         <- op(nodeop),
            with(node,op(nodeop)),
            (equiv)(node),
            not(and(with(node,kid(n,nkid)),
                    kid(n,not((same)(nkid)))))),
     (same)(node);
```

Figure 7: Parameterized version of sametree

of import lists, both to define a recursive anonymous predicate, and to make *equiv* available at once to all evaluations of that predicate. Given that definition, the following

```
sametree(node,
    FN((n) <- if(aconst(c),
                 with(n, aconst(c)),
                 and());))
```

would then test whether the current tree has the same structure as underneath node *and* such that all corresponding constants are the same.

3.2 Queries as Objects

Sometimes one wishes to build a collection or some other kind of aggregate of all objects found by a query. Unfortunately, when backtracking to get to the next hit, information about the previous hit will generally be lost. One solution is to rewrite the query into a conjunctive form, as we did in the previous section converting writing `flatten` as a conjunctive version of `somenode` (see Figure 5). We can already see that even in simple cases this process can be non-trivial and is not readily generalized.

It may also be the case for some conjunctive queries that they require memory proportional to the size of the data structure being searched, instead of merely memory proportional to the *depth* of the data structure. Judicious use of `if()` — ASTLOG's moral equivalent of the cut operator — can avoid this, but this is sometimes cumbersome to get right.

As it happens, Prolog provides a number of *set-predicates* for accumulating query results. For example,

`bagof(x,` *term*`, list)`

binds `list` to a list of the bindings of `x` corresponding to each instance where *term*holds true. Unfortunately, this is usually implemented in terms of `assert` and `retract`, meaning we would have to abandon the idea

- `query(`*fterm, query-pred*`)`
  The embedded query state object created from *fterm* satisfies *query-pred*.
- `qnext(`*pred, thisquery-pred, nextquery-pred*`)`
  If the current embedded query state is a failure, *pred* is true, otherwise the current object satisfies *thisquery-pred* and, after the embedded query is advanced to the next hit or to failure, the resulting query state satisfies *nextquery-pred*.
- `qget(`*object-pred, ...*`)`
  Each *object-pred* matches the object bound to the corresponding variable in the head of the embedded query corresponding to the current query state object. An error will be raised if the embedded query has failed or if any head variable is not bound to an object.

Figure 8: Embedded Query State Primitives

of keeping our script small and fixed. Even just adding this as a new primitive is dubious if we have to add, say, another new primitive to merely count query hits, and yet more new primitives for each accumulation method anyone dreams up.

The key observation is that the execution model of ASTLOG allows for the possibility of treating some subset of its own internal structures as "external" objects which can then serve as the current object of various kinds of queries. To be sure, some care needs to be exercised, since the internal structures of ASTLOG are not static the way the program ASTs are. We can however, take a query whose hits we wish to accumulate, and encapsulate its state after a given hit as an ASTLOG object. Such an *embedded* query in a given state can now be the current object for the evaluation of some other predicate term. We thus only need to provide suitable primitive predicates applicable to query-state objects that may be used in such a term. Figure 8 lists these primitives.

Using this mechanism, it is then possible to define a wide variety of accumulators of query results. Given an AST node, and a query to see if there exists a descendant satisfying `test(x)`

```
() <- somenode(test(x));
```

the corresponding query to count the number of descendants satisfying `test(x)` would be

```
(n)  <-  query(FN(() <- somenode(test(x)); ),
                qcount(n));
```

where `qcount/1` is defined as in Figure 9. Evaluating the `query()` term starts an embedded query corresponding to the first operand and builds a query state object representing the resulting first state (first hit

```
qcount(n) <- qcount(0, n);

qcount(sofar, return)
  <- qnext(unify(sofar, return),
           with(sofar, minus(sofarp1,1)),
           qcount(sofarp1, return));

qlist(lst)
  <- qnext(unify(lst, []),
           qget(first(lst,rest)),
           qlist(rest));

// utilities
first([o|rest],rest) <- o;
unify(x,x);
```

Figure 9: Query Accumulators `qcount` and `qlist`

or failure). This object then becomes the current object to which we try to match `qcount(n)`. It is the `qnext()` term therein that does the actual work. If the query-state is a success state, we increment the count of hits thus far (`sofar`), advance the embedded query, and recursively try to match a `qcount` term to the new state. If the query-state is a failure, we unify the count of hits thus far with the return variable.

To build a list of bindings for `x` corresponding to the query hits, we can do

```
(list)  <-  query(FN((x) <- somenode(test(x)); ),
                   qlist(list));
```

which is essentially the same as before except that now `qlist(list)` uses `qget` to examine the query state. Since the embedded query has only one head variable `x`, the `qget` term must likewise have at most one operand.

Some care is required when using embedded queries to phrase them so that the head variables will always be bound to objects. `qget()` will in fact raise an error if a head variable is not bound to an object. This requirement is crucial since, with a non-object term, there is no guarantee that said term will remain intact when the embedded query backtracks to the next state. Better to keep terms constructed by an embedded query from polluting the outer world.

The mechanism is also somewhat impure in that evaluating a `qnext` on a given query state object essentially destroys that object. Subsequent attempts to match additional terms against that query state will raise an error since the state of a query is lost once we advance it.

4 Implementation

ASTLOG has been implemented as an interpreter in roughly 11,000 lines of C++ for the core ASTLOG interpreter and supporting utilities. Another 1100 lines define the roughly 60 primitives and supporting structures to invoke the various functions of the AST library. Coverage of the library API is in not entirely complete, but it is sufficient to perform various interesting tasks:

- Finding all instances of a simple assignment expression (=) occurring in *any* boolean context, for example,

  ```
  if ((major == SORTM)
      || (major == MEMORYM)
      || ((major == BUFFERM)
          && (minor = B_NOIO)))
  ```

- Finding all instances of an equality-test (==) or dereference expression occurring in *any* void context (i. e., where results are discarded); the converse to the previous problem.

- Finding all `case` statement fall-throughs, i. e., where the preceding statement is not a break.

- Finding various patterns of irreducible control-flow in functions.

- Obtaining all static call-graph edges.

- Computing the McCabe cyclomatic complexity [McC76] of a function. Our code to do so looks like

  ```
  mccabe(n) <- query(
     FN(()<- somenode(
                op(or(#IF,#FOR,#DO,
                      #WHILE,#CASE,
                      #?,#||,#&&)));)
     qcount(minus(n,1))
  );
  ```

  which might be compared with the 17-line version in Aria [DR96]. Admittedly, fairness would probably entail including the definitions of `somenode` and `qcount` as well.

- Finding gaps (unused space due to alignment rules) in structure definitions; this is a matter of traversing C type structures rather than ASTs.

A typical running time (on a 200MHz Pentium P6 with 64meg of RAM) for a one-pass search that evaluates a simple predicate on every AST node in Microsoft SQLserver (roughly 450,000 lines, 4300 functions) is roughly 10 minutes, of which 7.5 minutes are taken up by the AST library building the actual trees. For Microsoft Word (roughly 2,000,000 lines) the corresponding times are 45-60 minutes of which about 30 minutes is taken up by the tree builder.

Though this dreadfully slow in comparison with `grep`, these times are arguably acceptable in comparison with the times taken by the actual compiler — what one might expect for a tool that requires the use of compiler's data structures. One is, of course, free to write arbitrarily non-linear programs in ASTLOG, so there are no guarantees. In any case we would doubtless see a certain amount of speedup if we actually were to attempt some kind of compilation of the ASTLOG code.

5 Conclusions and Future Work

We have described a language for doing syntax-level analysis for C/C++ programs, though the core language is, in fact, adaptable to many other kinds of structures. As with previous such tools, the utility to users who are thus no longer required to write their own parse/semantic-analysis phase is apparent. The contribution here is a pattern language sufficiently powerful to provide traversal possibilites beyond what is naturally available in prior `awk`-like frameworks while avoiding some of the inefficiencies of importing the entire program structure into a logical inference engine. The Pan work [BGV90] stressed the need to partition code and data; this we have done in a rather straightforward way. The surprise is that the Prolog-with-an-ambient-current-object model turns out to be so well suited to analyzing treelike structures.

To be sure, there are various rough edges:

1. As already noted, embedded queries are slightly unsafe; there may exist a more robust set of primitives to use. Some form of type inference to detect unsafe uses of `qnext` may also be worth considering. More generally, there is the issue of typing of ASTLOG expressions to reduce the incidence of unbound variables or objects of the wrong type appearing as operands where object-references of a particular type are required.

2. Occasionally, we run up against the generally cumbersome nature of arithmetic in Prolog, which is arguably *worse* in ASTLOG. The "inside-out functional" nature of ASTLOG may be good for AST patterns, but it can make arithmetic operations like

   $$\texttt{with(n,divide(minus(x,1),2))}$$

   downright unreadable. Algebraic syntax could help, e. g.,

   $$\texttt{with}(\texttt{n}, (\texttt{x} - 1)/2)$$

   but even so, one must stare at this pretty hard to realize that `n` is being multiplied by 2 and then incremented by 1.

   One possibility is to complicate the language by introducing actual "forward" functional operator definitions. For example, with such forward operators for addition and multiplication, one could then write

   $$\texttt{with}(2 * \texttt{n} + 1, \texttt{x})$$

   where the appearance of the + (`plus`) term in a slot normally requiring an object reference invokes the forward return-value-to-context definition of the operator + to sum its operands rather than the usual "backward" return-value-to-operand definition (see Figure 2) in which one operand is treated as a predicate.

3. Though there is a surprising amount of mileage to be had via instantiating terms with unbound variables in them, there are those occasions when a genuinely mutable data structure is required. Fortunately, given the strong partition between the script/database and the objects, having mutable objects exist and primitives that side-effect them when they match would not disrupt ASTLOG's execution model.

4. Currently, new primitives need to be manually written. Given the current collection of macros available, this is not actually an arduous task. Still, while language-independence was not one of our priorities, given that the core language is rather language-independent anyway, one would hope for a more automatic means of adapting ASTLOG to work with other language parsers, perhaps by adapting GENII [Dev92] or some similar tool to generate code for the basic primitive predicate operators for a fresh language.

6 Acknowledgements

ASTLOG would not have been possible without the existence of an AST library for C/C++ implemented by the members of Program Analysis group at Microsoft Research, particularly Linda O'Gara, David Gay, Erik Ruf and Bjarne Steensgaarde. I would also like to thank Bruce Duba, Michael Ernst, Chris Ramming, and the conference reviewers for much useful commentary and discussion.

References

[AKW86] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison Wesley, Reading, MA, 1986.

[BCD+88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, MA, 1988.

[BGV90] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The pan language-based editing system for integrated development environments. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 77–93, Irvine, CA, 1990.

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the Fourteenth International ACM Conference on Software Engineering*, pages 138–156, 1992.

[Dev92] Premkumar T. Devanbu. Genoa - a customizable, language-and-front-end independent code analyzer. In *Proceedings of the Fourteenth International ACM Conference on Software Engineering*, pages 307–319. ACM Press, 1992.

[DR96] Premkumar T. Devanbu and David S. Rosenblum. Generating testing and analysis tools with aria. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

[GA96] William G. Griswold and Darren C. Atkinson. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the IEEE Workshop on Program Comprehension*. ACM Press, 1996.

[LR95] David A. Ladd and J. Christopher Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.

[McC76] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press series in logic programming. The MIT Press, Cambridge, MA, 1986.

Appendix

For those who would prefer to see a slightly more formal description, we include a brief outline of an operational semantics for ASTLOG in Figure 10, one that bears some resemblance to the actual implementation.

For any given term that is not an object reference, one may imagine there being numerous instances of that term in existence at any given time. We differentiate the various instances by assigning each a unique *frame identifier* (f) which is only significant for its identity. A variable v occurring within a given term t may, for a particular instance $\langle f, [\![t]\!]\rangle$ of that term, be bound to some object o or other term instance $\langle f', [\![t']\!]\rangle$, this being indicated by having a *binding*, i.e., one of $\langle f, [\![\mathrm{v}]\!]\rangle \sim o$ or $\langle f, [\![\mathrm{v}]\!]\rangle \sim \langle f', [\![t']\!]\rangle$ present in the current *binding stack*, which in turn is nothing more than a list of bindings. The semantic function $\mathrm{vlookup}(B, \langle f, [\![t]\!]\rangle)$ returns

- $\langle f, [\![t]\!]\rangle$ itself if t is not a variable.
- $\perp$ if the variable t is not bound in B.
- o if $\langle f, [\![t]\!]\rangle \sim o$ is in B
- $\mathrm{vlookup}(B, \langle f', [\![t']\!]\rangle)$ if $\langle f, [\![t]\!]\rangle \sim \langle f', [\![t']\!]\rangle$ is in B.

At any given time, the full state of our abstract machine is described by a *failure* of the form $B \vdash C :: F$ which consists of

- the current binding stack B,
- the current *continuation* $C = (o, f, g, C')$, which in turn consists of a current object o, a current frame identifier f, a current *goal*, usually a term, but this can also be one of the auxiliary goals "apply(...)" or "cut(...)," and finally another continuation C' to which we advance if the goal succeeds
- the next failure F, to which we advance if the current goal fails.

Note that unlike the case where the goal succeeds, failure may involve undoing one or more bindings; thus, a failure (F) contains its own binding stack (a subset of B) whereas the continuations (C, C') do not.

The bottom half of Figure 10 (partially) defines a transition relation between states of the abstract machine. Given an initial current object o and a query $[\![query]\!]$ with n head variables, we take the initial state to be

$$F_0 = [] \vdash (o, f_0, \mathrm{apply}(f_0, [\![query]\!], [\![\mathrm{v}_1, \ldots, \mathrm{v}_n]\!]), \mathrm{yes}) :: \mathrm{no}$$

If there is a sequence of transitions

$$F_0 \longrightarrow^* B_1 \vdash \mathrm{yes} :: F_1$$

then we have a hit and the various query head bindings are available as vlookup$(B_1, \langle f_0, [\![\mathtt{v}_i]\!]\rangle)$ for $i = 1 \ldots n$. Likewise, if

$$F_k \longrightarrow^* B_k \vdash \text{yes} :: F_{k+1}$$

then we have a $(k+1)^{\text{th}}$ hit.

The semantic function

$$\text{mgu}(B, f, [\![t_1, \ldots, t_n]\!], f', [\![t'_1, \ldots, t'_n]\!])$$

returns an augmented binding stack that includes B together with those additional bindings that make up the most general unifier of the respective term instances $\langle f, [\![t_1]\!]\rangle$ with $\langle f', [\![t'_1]\!]\rangle$, etc.... If there is no most general unifier, mgu() returns ufail.

In the actual implementation, because the script is fixed, we may precompute at load time mgus of all pairs of same-operator-and-arity compound terms occurring in the script, making clause invocation no more expensive than a function call in many cases. We also omit the "occurs check" [SS86] for the run-time portion of unification (i.e., where we're transitively following variable bindings), with the usual increase in speed and infinite-loop risk. Thus far, unification has played a somewhat smaller role in ASTLOG scripts than expected, so there's some question whether we need to be doing even this much.

As noted above objects only unify with equal objects. The idea of allowing an object to unify with a compound predicate term that matches it has been considered, but rejected due to the significant complications it would introduce. Also, once one has subgoals being attempted during the course of unification, the user's control over evaluation order is drastically reduced, something to be avoided if one is interested in having users being able to write efficient scripts.

$$\begin{array}{rll}
\mathit{CompTerms} = & \mathit{OpTerms} + \mathit{LambdaTerms} + \mathit{AppTerms} & \llbracket \texttt{op}(t_1, \ldots) \rrbracket, \llbracket \texttt{FN}(\mathit{clauses}) \rrbracket, \llbracket (\mathit{fterm})(t_1, \ldots) \rrbracket \\
\mathit{NonObjTerms} = & \mathit{Vars} + \mathit{CompTerms} & \llbracket \texttt{var} \rrbracket \\
\mathit{Terms} = & \mathit{NonObjTerms} + \mathit{Objects} & \llbracket o \rrbracket \\
\mathit{Goals} = & \mathit{Terms} & \llbracket t \rrbracket \\
& + \mathit{FrameIds} \times \mathit{LambdaTerms} \times \mathit{Terms}^* & \mathrm{apply}(f, \llbracket \mathit{fterm} \rrbracket, \llbracket t_1, \ldots \rrbracket) \\
& + \mathit{Failures} \times \mathit{Terms} & \mathrm{cut}(F, \llbracket t \rrbracket)
\end{array}$$

$$\begin{array}{rll}
\mathit{Objects} & & o \\
\mathit{FrameIds} & & f \\
\mathit{Bindings} = & (\mathit{FrameIds} \times \mathit{Vars}) & \langle f, \llbracket \mathtt{v} \rrbracket \rangle \sim o \\
& \times (\mathit{Objects} + (\mathit{FrameIds} \times \mathit{NonObjTerms}))) & \langle f, \llbracket \mathtt{v} \rrbracket \rangle \sim \langle f', \llbracket t \rrbracket \rangle \\
\mathit{BindingStacks} = & \mathit{Bindings}^* & B \\
\mathit{Conts} = & (\mathit{Objects} \times \mathit{FrameIds} \times \mathit{Goals} \times \mathit{Conts}) + \{\mathrm{yes}\} & (o, f, \llbracket t \rrbracket, C) \\
\mathit{Failures} = & ((\mathit{BindingStacks} + \{\mathrm{ufail}\}) \times \mathit{Conts} \times \mathit{Failures}) & B \vdash C :: F \\
& + \{\mathrm{no}\} &
\end{array}$$

$$\begin{array}{rl}
\mathrm{vlookup}: & \mathit{BindingStacks} \times \mathit{FrameIds} \times \mathit{NonObjTerms} \rightarrow \{\perp\} + \mathit{Objects} + (\mathit{FrameIds} \times \mathit{NonObjTerms}) \\
\mathrm{flookup}: & \mathit{OpIds} \times N \rightarrow \{\perp\} + \mathit{LambdaTerms} \\
\mathrm{frames}: & \mathit{BindingStacks} \rightarrow \mathcal{P}(\mathit{FrameIds}) \\
\mathrm{mgu}: & \mathit{BindingStacks} \times \mathit{FrameIds} \times \mathit{Terms}^* \times \mathit{FrameIds} \times \mathit{Terms}^* \rightarrow \mathit{BindingStacks} + \{\mathrm{ufail}\}
\end{array}$$

$$B \vdash (o, f, \llbracket o \rrbracket, C) :: F \longrightarrow B \vdash C :: F$$

$$\frac{o \neq o'}{B \vdash (o, f, \llbracket o' \rrbracket, C) :: F \longrightarrow F}$$

$$\frac{\mathrm{vlookup}(B, \langle f, \llbracket \texttt{var} \rrbracket \rangle) = \langle f', \llbracket \mathit{term} \rrbracket \rangle}{B \vdash (o, f, \llbracket \texttt{var} \rrbracket, C) :: F \longrightarrow B \vdash (o, f', \llbracket \mathit{term} \rrbracket, C) :: F}$$

$$\frac{\mathrm{vlookup}(B, \langle f, \llbracket \texttt{var} \rrbracket \rangle) = \perp}{B \vdash (o, f, \llbracket \texttt{var} \rrbracket, C) :: F \longrightarrow [@B, \langle f, \llbracket \texttt{var} \rrbracket \rangle \sim o] \vdash C :: F}$$

$$\frac{\mathrm{vlookup}(B, \langle f, \llbracket t_1 \rrbracket \rangle) = o'}{B \vdash (o, f, \llbracket \texttt{with}(t_1, t_2) \rrbracket, C) :: F \longrightarrow B \vdash (o', f, \llbracket t_2 \rrbracket, C) :: F}$$

$$\frac{\mathrm{flookup}(\llbracket \texttt{op} \rrbracket, n) = \mathit{fterm}, \mathit{fterm} \neq \perp}{B \vdash (o, f, \llbracket \texttt{op}(a_1, \ldots, a_n) \rrbracket, C) :: F \longrightarrow B \vdash (o, f, \mathrm{apply}(f, \llbracket \mathit{fterm} \rrbracket, \llbracket a_1, \ldots, a_n \rrbracket), C) :: F}$$

$$\frac{\mathrm{vlookup}(B, \langle f, \llbracket \mathit{fterm} \rrbracket \rangle) = \langle f', \llbracket \mathit{fterm}' \rrbracket \rangle}{B \vdash (o, f, \llbracket (\mathit{fterm})(a_1, \ldots, a_n) \rrbracket, C) :: F \longrightarrow B \vdash (o, f, \mathrm{apply}(f', \llbracket \mathit{fterm}' \rrbracket, \llbracket a_1, \ldots, a_n \rrbracket), C) :: F}$$

$$B \vdash (o, f, \mathrm{apply}(f', \llbracket \texttt{FN}\{\mathtt{i}_1, \ldots\}() \rrbracket, \llbracket a_1, \ldots, a_n \rrbracket), C) :: F \longrightarrow F$$

$$\frac{f'' \notin \mathrm{frames}(B), B' = \mathrm{mgu}([@B, \langle f'', \llbracket \mathtt{i}_1 \rrbracket \rangle \sim \langle f', \llbracket \mathtt{i}_1 \rrbracket \rangle, \ldots], f, \llbracket a_1, \ldots, a_n \rrbracket, f'', \llbracket t_1, \ldots, t_n \rrbracket)}{\begin{array}{l} B \vdash (o, f, \mathrm{apply}(f', \llbracket \texttt{FN}\{\mathtt{i}_1, \ldots\}((t_1, \ldots, t_n)\mathit{body}_1 \ldots; \mathit{clause}_2 \ldots) \rrbracket, \llbracket a_1, \ldots, a_n \rrbracket), C) :: F \\ \qquad \longrightarrow B' \vdash (o, f'', \llbracket \texttt{and}(\mathit{body}_1 \ldots) \rrbracket, C) \\ \qquad\qquad :: (B \vdash (o, f, \mathrm{apply}(f', \llbracket \texttt{FN}\{\mathtt{i}_1, \ldots\}(\mathit{clause}_2 \ldots) \rrbracket, \llbracket a_1, \ldots, a_n \rrbracket), C) :: F) \end{array}}$$

$$\mathrm{ufail} \vdash (o, f, \llbracket t \rrbracket, C) :: F \longrightarrow F$$

$$\begin{array}{r}
B \vdash (o, f, \llbracket \texttt{and}(t_1, t_2) \rrbracket, C) :: F \longrightarrow B \vdash (o, f, \llbracket t_1 \rrbracket, (o, f, \llbracket t_2 \rrbracket, C)) :: F \\
B \vdash (o, f, \llbracket \texttt{or}(t_1, t_2) \rrbracket, C) :: F \longrightarrow B \vdash (o, f, \llbracket t_1 \rrbracket, C) :: (B \vdash (o, f, \llbracket t_2 \rrbracket, C) :: F) \\
B \vdash (o, f, \llbracket \texttt{if}(t_1, t_2, t_3) \rrbracket, C) :: F \longrightarrow B \vdash (o, f, \llbracket t_1 \rrbracket, (o, f, \mathrm{cut}(F, \llbracket t_2 \rrbracket), C)) :: (B \vdash (o, f, \llbracket t_3 \rrbracket, C) :: F) \\
B \vdash (o, f, \mathrm{cut}(F', \llbracket t \rrbracket), C) :: F \longrightarrow B \vdash (o, f, \llbracket t \rrbracket, C) :: F'
\end{array}$$

Figure 10: Outline of ASTLOG Operational Semantics

KHEPERA: A System for Rapid Implementation of Domain Specific Languages

Rickard E. Faith   Lars S. Nyland   Jan F. Prins

Department of Computer Science
University of North Carolina
CB #3175, Sitterson Hall
Chapel Hill NC 27599-3175
{faith,nyland,prins}@cs.unc.edu

Abstract

The KHEPERA system is a toolkit for the rapid implementation and long-term maintenance of domain specific languages (DSLs). Our viewpoint is that DSLs are most easily implemented via source-to-source translation from the DSL into another language and that this translation should be based on simple parsing, sophisticated tree-based analysis and manipulation, and source generation using pretty-printing techniques. KHEPERA emphasizes the use of familiar, pre-existing tools and provides support for transformation replay and debugging for the DSL processor and end-user programs. In this paper, we present an overview of our approach, including implementation details and a short example.

1 Introduction

Domain specific languages (DSL) can often be implemented as a source-to-source translator composed with a processor for another language. For example, PIC [8], a classic "little language" for typesetting figures, is translated into `troff`, a general-purpose typesetting language. Language composition can be extended in either direction: the CHEM language [1], a DSL used for drawing chemical structures, is translated into PIC, while `troff` is commonly translated into PostScript.

Other DSLs translate into general-purpose high-level programming languages. For example, ControlH, a DSL for the domain of real-time Guidance, Navigation, and Control (GN&C) software, translates into Ada [5]; and RISLA, a DSL for financial engineering, translates into COBOL [18].

The composition of a DSL processor with (for example) a C compiler is attractive because it provides portability over a large class of architectures, while achieving performance through the near universal availability of architecture-specific optimizing C compilers.

Yet there are some drawbacks to this approach. While DSLs are often simpler than general purpose programming languages, the domain-specific information available may result in a generated program that can be much larger and substantially different in structure than the original code written in the DSL. This can make debugging very difficult: an exception raised on some line of an incomprehensible C program generated by the DSL processor is a long way removed in abstraction from the DSL input program.

Since the DSL processor is composed with a native high-level compiler, and does not have to perform machine-code generation or optimization, we believe that there are some basic differences between the construction of a compiler for a general purpose programming language and the construction of a translator for a DSL. Our view is that DSL translation is most simply expressed as

1. simple parsing of input into an abstract syntax tree (AST),
2. translation via sophisticated tree-based analysis and manipulation, and
3. output source generation using versatile pretty-printing techniques.

We add the additional caveat that the translation process retain enough information to support the inverse mapping problem, i.e., given a locus in the output source, determine the tree manipulations and input source elements that are responsible for it. This

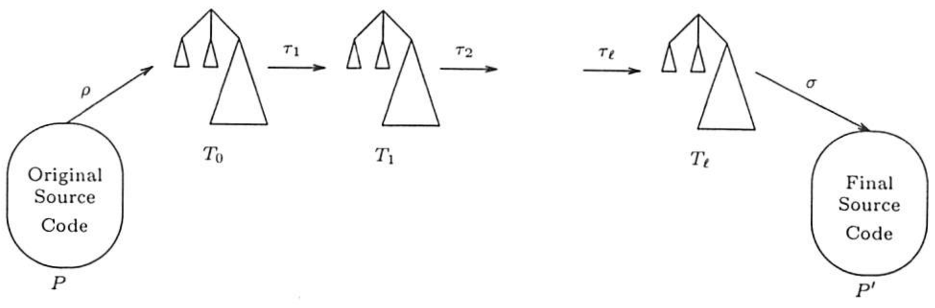


Figure 1: Transformation Process

facility would be useful both for the DSL developer to trace erroneous translation and for the DSL user to trace (run-time) errors back to the input source.

For the translation step we advocate the use of arbitrary AST traversals and transformations. We believe that this approach is simpler for source-to-source translation than the use of attribute grammars, since it decouples the AST analysis and program synthesis from the grammar of the input and output languages. Further, this approach minimizes the need for parsing "heroics", since simple grammars, close or identical to the natural specification of the DSL syntax, can be used to generate an AST that is specialized in subsequent analysis. By decoupling the input grammar, translation process, and output grammar, this approach is better able to accommodate changes during the evolution of the DSL syntax and semantics.

Throughout this paper, we will use "AST" to refer to abstract syntax tree derived from parsing the input file, and to any intermediate tree-based representations derived from this original AST, even if those representations do not strictly represent an "abstract syntax".

In our own work we use the DSL paradigm in the compilation of parallel programs. We are particularly interested in the translation into HPF of irregular computations expressed in the PROTEUS [12] language, a DSL providing specialized notation. Our observation was that we were spending a disproportionate amount of effort working on a custom translator implementation to incorporate changes in PROTEUS syntax and improvements in the translation scheme—thus we were motivated to investigate general tool support for DSL translation to simplify this process.

1.1 Goals for a DSL Implementation Toolkit

The implementation of a DSL translator can require considerable overhead, both for the initial implementation and as the DSL evolves. A toolkit should leverage existing, familiar tools as much as possible. Use of such tools takes advantage of previous implementor knowledge and the availability of comprehensive resources explaining these tools (which may not be widely available for a DSL toolkit).

A transformational model for DSL design fits in well with these high-level goals. Consider the problem of translating a program, P, written in the domain specific language, L. In Figure 1, T_0 is an AST which represents P after the parsing phase, ρ. T_ℓ is the final transformed AST, and P' is a valid program in the output language, L', constructed from T_ℓ during the pretty-printing phase, σ. The transformation process is viewed as a sequential application of various transformations functions, $\tau_{k+1}(T_k) = T_{k+1}$, to the AST. The determination of which transformation function to apply next may require extensive analysis of the AST. Once the transformation functions are determined, however, they can be rapidly applied for replay or debugging.

Within a transformational model, a DSL-building toolkit can simplify the implementation process by providing specialized tools where pre-existing tools are not already available, and to transparently integrate support for debugging within this framework.

The KHEPERA system facilitates both the problem of rapid DSL prototyping and the problem of long-term DSL maintenance through the following specific design goals:

Familiar, modularized parsing components. KHEPERA supports the use of familiar scanning and parsing tools (e.g., the traditional `lex` and `yacc`, or the newer PCCTS [11]) for implementation of a DSL processor. Because KHEPERA concentrates on providing the "missing pieces" that help with rapid implementation of DSLs, previous knowledge can be utilized, thereby decreasing the slope of the learning curve necessary for the rapid implementation of a DSL.

Familiar, flexible, and efficient semantic analysis. KHEPERA uses the source-to-source transformational model outlined in Figure 1. This model uses tree-pattern matching for AST manipulation, analysis, and attribute calculation. For tedious but common tasks, such as tree-pattern matching, sub-tree creation, and sub-tree replacement, KHEPERA provides a little language for describing tree matches and for building trees. For unpredictable or language-specific tasks, such as attribute manipulation or analysis, the KHEPERA little language provides an escape to a familiar general-purpose programming language (C). Standard tree traversal algorithms are supported (e.g., bottom up, top down), as well as arbitrarily complicated syntax-directed sequencing. Rapid pattern matching is provided via data-structure maintenance, which can perform rapid pattern matches in a standard tree traversal order for many commonly-used patterns.

Familiar output mechanism. A pretty-printing facility is provided that can output the AST in an easily readable format at any time. One strong advantage of this pretty-printer when compared with other systems is that it will always be able to print the AST, regardless of how much of the transformation has been performed. If the AST is in the original input format or the original output format, then the pretty-printed program will probably be executable in the input language, L, or the output language, L'. However, if the AST being printed is one of the T_n intermediate trees, then the output will use some combination of the syntax of L and L', with a fallback to simple S-expressions for AST constructs which do not have well-defined concrete syntax. While the program printed may not be executable, it does use a familiar syntax which may be helpful for the human when replaying transformations while debugging.

Debugging support for DSL translation. KHEPERA tracks transformation application and AST modifications, can replay the transformation sequence, and has support for answering questions about which transformations were applied at which points on the AST. This is helpful when writing and debugging the DSL processor, as well as when implementing a debugger for the DSL itself.

Transformations are either written in the high-level KHEPERA language and are transformed by KHEPERA into executable C with calls to the KHEPERA library (as discussed in Section 4.6 and shown in Figure 8 and Figure 9); or the transformations are written using explicit calls to the KHEPERA library tree manipulation functions. In either case, low-level hooks in the KHEPERA library track debugging information when nodes or subtrees are created, destroyed, copied, or replaced. This low-level information can be analyzed to provide the ability to navigate through intermediate versions of the transformed program, and the ability to answer specific queries that support the debugging of the final transformed output:

- setting breakpoints
- determining current execution location (e.g., in response to a breakpoint or program exception)
- reporting a procedure traceback
- displaying values of variables

These tracking and debugging capabilities are the subject of Faith's forthcoming dissertation and will be not be discussed in detail in this paper. An example of setting a breakpoint will be shown in Section 4.

2 Related Work

KHEPERA is similar to some compiler construction kits. However, these systems usually restrict the scanning and parsing tools used [6]; specify AST transformations using a low-level language, such as C [17] (instead of a high-level transformation-oriented language); or require that the AST always conforms to a single grammar specification, making translation from one language to another difficult [4, 3, 14]. Further, some systems rely on an attribute grammars for all AST transformations, without providing for a more general-purpose scheme for tree-pattern matching and replacement.

SORCERER, from the PCCTS toolkit [11], is the most similar, since it does not require the use of specific scanning and parsing tools, and since it provides a "little language" in the style of `lex` and `yacc` with embedded procedures written in another general-purpose programming language (e.g., C). SORCERER and KHEPERA share abilities to describe tree structures, perform syntax-directed translations, and support the writing of AST-based interpreters. In contrast, KHEPERA also supports rule-based translations that do not require a complete grammar specification; KHEPERA rules are well suited for the construction of "use-def" chains, data-flow dependency graphs, and other compiler-required analyses; and writing pretty-printer rules in KHEPERA does not require a complete tree-grammar specification. This allows pretty-printing to easily take place during grammar evolution.

None of the previous systems, including SORCERER, contain built-in support for "replay" of transformations, or for automatic and transparent tracking of debugging information. When translating programs from one language to another, the "discovery" of the best order for transformation application is often difficult, involving considerable AST analysis. The code to perform this analysis is often difficult to verify or is undergoing constant change during the implementation phase of a DSL. However, after the transformations are discovered and recorded in a database, a much simpler program (i.e., one that is easier to verify) could be written that applies all of the discovered transformations in the specified order, thereby proving, by construction, that the translation preserves semantics. In this case, only the semantics preserving characteristics of the transformations themselves must be proven—not the code which performs analysis and discovery. While we have not yet implemented such a prover, we have utilized the transformation discovery and replay capabilities of KHEPERA to implement a browser that presents intermediate views of the transformation process, and which can answer typical queries posed by a debugger (see Section 4.6).

3 Overview of KHEPERA

The KHEPERA library provides low-level support for:

- building an AST
- applying transformation rules to the AST (tree traversal, matching, and replacement)
- "pretty-printing" the P' source code from the T_ℓ AST (pretty-printing is actually the σ "transformation")

An overview of the KHEPERA system is shown in Figure 2. KHEPERA encapsulates low-level details of the DSL implementation: AST manipulation, symbol and type table management, and management of line-number and lexical information. On a higher level, library routines are available to support pretty-printing (currently, with a small language to describe how to print each node type in the AST), type inference, and the tracking functions for debugging information. Further, a "little language" has been implemented to support a high-level description of the transformation rules. If transformation rules are written in the KHEPERA language, or if they are written in an ad hoc manner using the underlying KHEPERA AST manipulation library, then the debugging tracking and transformation replay support will be automatically provided.

An overview of how the KHEPERA system fits into a complete DSL implementation solution is shown in Figure 3. In the example shown in the next section, we explain how the scanner and parser specifications are simplified by using calls to the KHEPERA library and will provide an example showing how other important input files are specified.

In Figure 4, the "DSL Processor" from from Figure 3 is expanded, showing the basic blocks that are created from the source code and showing how the DSL processor is used during the compilation of a program written in the DSL.

4 Example

A simple language translation problem based on [12] will be used to illustrate the KHEPERA system. The DSL is a subset of Fortran 90 with the addition of a *sequence comprehension* construct that can be used to construct (nested) sequences. The translation problem is to remove all sequence comprehension constructs and replace them with simple data-parallel operations, yielding a program suitable for compilation with a standard Fortran 90 compiler.

4.1 Example DSL Syntax

The lexical elements of the DSL are:

Id **Num** `(/ /) ( ) + , : = in`

A program is described by the following context-free grammar (CFG):

program ::= *statement-list*

statement ::= **Id** = *expression*

statement-list ::= *statement*

| *statement-list statement*

expr ::= **Id**

| **Num**

| *expr* + *expr*

| `length(` *expr* `)`

| `range(` *expr* `)`

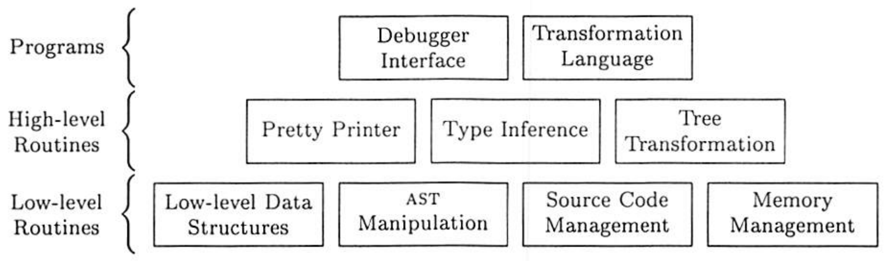


Figure 2: The KHEPERA Transformation System

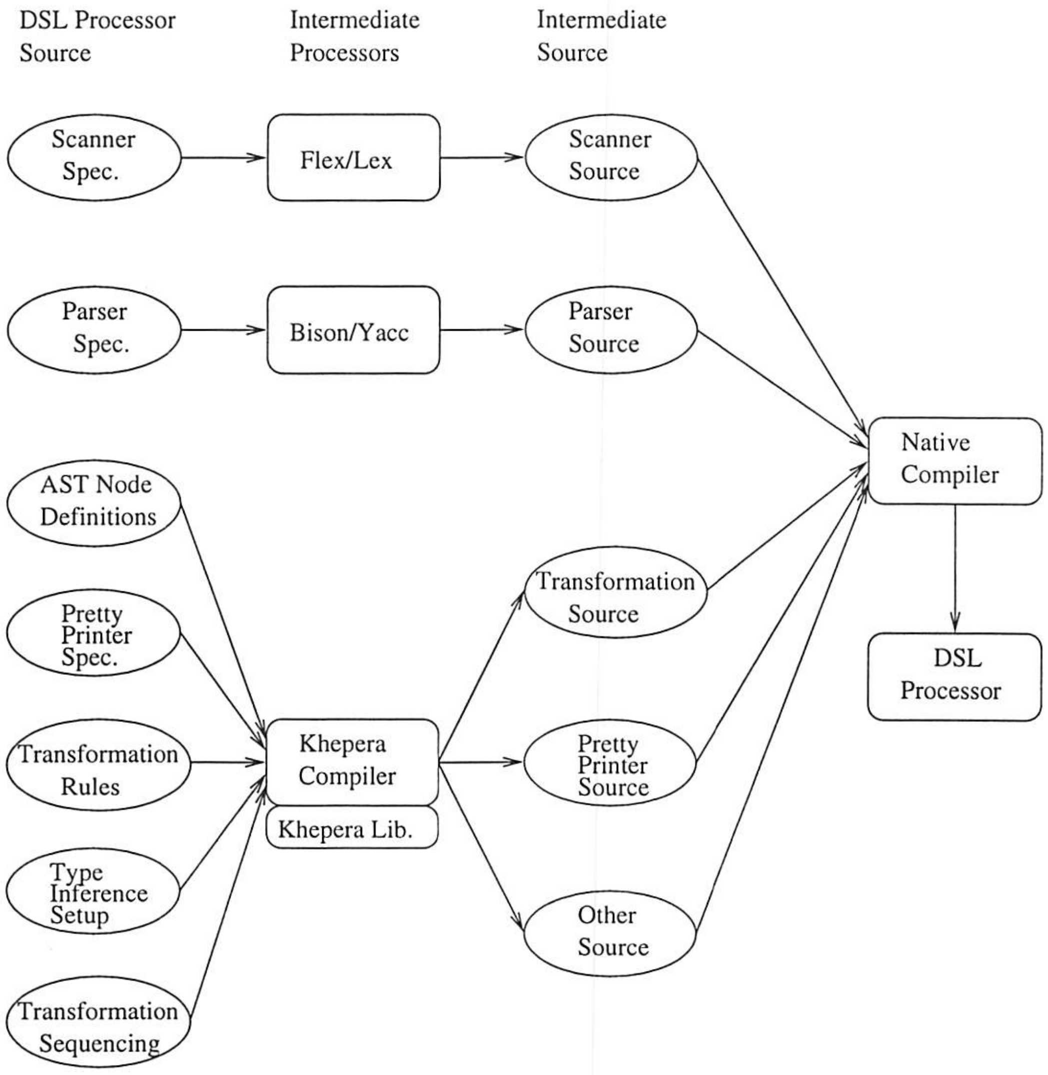


Figure 3: Using the KHEPERA Transformation System

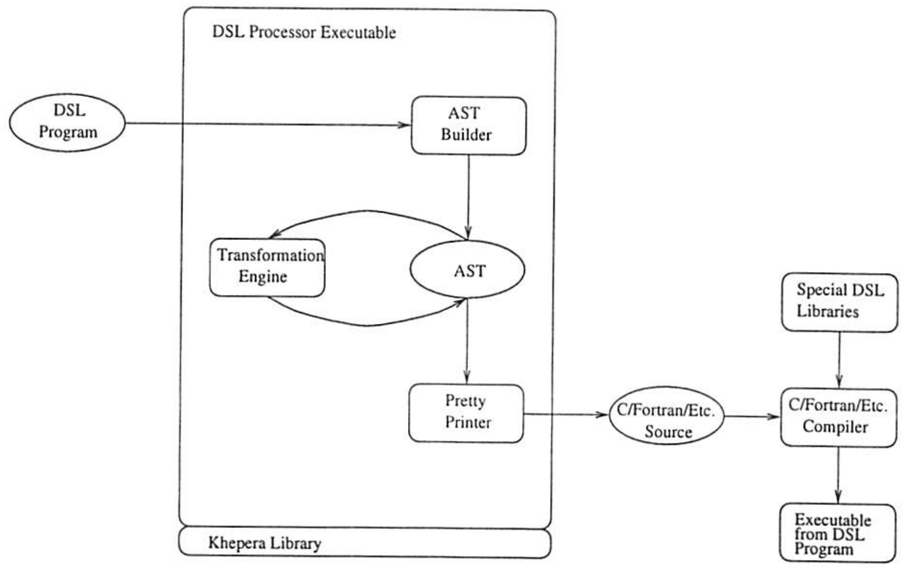


Figure 4: Using the DSL Processor

| `dist(` *expr* `,` *expr* `)`

| `(/` *expr-list* `/)`

| `(/` **Id** `in` *expr* `:` *expr* `/)`

For this example, we use the array constructor notation from Fortran 90 to specify literal sequences and a similar notation to specify the sequence comprehension construct. However, the sequence comprehension construct creates arbitrarily nested, *irregular* sequences. (In contrast, the array constructor from Fortran 90 can only generate vectors or rectangular arrays.)

4.2 Example DSL Semantics

DSL values have types drawn from $D = \mathbf{Int}|\mathrm{Seq}(D)$. We define, $\forall n \in \mathbf{Int}, c \in D$:

$$\begin{aligned}\texttt{range}(n) &= (/\ 1, 2, \ldots, n\ /)\\ \texttt{dist}(c, n) &= (/\ c, c, \ldots, c\ /)\end{aligned}$$

with $\texttt{length}(\texttt{dist}(c,n)) = \texttt{length}(\texttt{range}(n)) = n$. For an expression, e, the sequence comprehension

$$(/\ i \texttt{ in } A : e(i)\ /)$$

yields the sequence of successive values of e obtained when i is bound to successive values in A.

For example, the sample program:

```
A = range(3);
B = (/ i in A: i + i /);
C = (/ i in A:
        (/ j in range(i): i /) /)
```

yields:

```
A = (/ 1, 2, 3 /)
B = (/ 2, 4, 6 /)
C = (/ (/ 1 /),
        (/ 2, 2 /),
        (/ 3, 3, 3 /) /)
```

We omit here a collection of type (inference) rules for the language that define a well-typed program.

4.3 Example Translation

We view a program in terms of the natural AST corresponding to the CFG of Section 4.1. In the AST, an application of one of the four basic operations is written as a function application node with the operation to be applied in the *name* attribute and a *depth* attribute that is 0. The children of the node are expression(s) for each of the arguments.

The following 3 rules can be used to eliminate all sequence comprehension constructs from the AST:

Rule 1

$$(/\ x_1 \texttt{ in } e_1 : x_1\ /) \longrightarrow e_1$$

Rule 2 Provided e_2 is an **Id** or **Num**, and $e_2 \neq x_1$,

$$\begin{array}{l} \texttt{(/}\ x_1\ \texttt{in}\ e_1\ \texttt{:}\ e_2\ \texttt{/)} \\ \qquad \longrightarrow \quad \texttt{dist(}\ e_2, \texttt{length(}\ e_1\texttt{))} \end{array}$$

Rule 3

$$\begin{array}{l} \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:} \\ \quad \texttt{fn\_app(}\ \texttt{name} = f, \\ \quad\quad\quad \texttt{depth} = d, \\ \quad\quad\quad \texttt{args} = n, \\ \quad\quad\quad e_1, \ldots, e_n\ \texttt{)}\ \texttt{/)} \\ \longrightarrow \quad \texttt{fn\_app(}\ \texttt{name} = f, \\ \quad\quad\quad \texttt{depth} = d + 1, \\ \quad\quad\quad \texttt{args} = n, \\ \quad\quad\quad \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:}\ e_1\ \texttt{/)}, \\ \quad\quad\quad \ldots, \\ \quad\quad\quad \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:}\ e_n\ \texttt{/)}\ \texttt{)} \end{array}$$

The resultant AST can be written out in as Fortran 90 with the depth attribute supplied as an extra argument to the basic functions (`add`, `length`, `range`, `dist`). Given an appropriate implementation of these basic four functions, the resultant program specifies fully parallel execution of each sequence comprehension construct, regardless of the degree of nesting and sequence sizes.

For example, using these rules, the program from Section 4.2 would be transformed as follows (using $f(\ldots)$ as a shorthand for $\texttt{fn\_app(name} = f, \ldots\texttt{)}$):

```
A = range(depth=0, 3)
B = add(depth=1, A, A)
C = dist(depth=1,
         A,
         length(depth=1,
                range(depth=1, A)))
```

Note that functions with $\texttt{depth} = 0$ operate on scalar arguments, whereas functions with $\texttt{depth} = 1$ operate on vector arguments.

The rules shown for this example are terminating and confluent. When the source language is more expressive and optimization becomes an issue, the rules used are not necessarily terminating, hence additional sequencing rules must be added to control rule application [10].

In the following sections, we shall show how KHEPERA can be used to implement translations, such as the one specified above, in an efficient manner.

4.4 Parsing and AST Construction

The AST is constructed using a scanner and parser generator of the implementor's choice with calls to the KHEPERA library AST construction routines. At the level of the scanner, KHEPERA provides support for source code line number and token offset tracking. This support is optional, but is very helpful for debugging. If the implementor desires line number and token offset tracking, the scanner must interact with KHEPERA in three ways: first, each line of source code must be registered. In versions of `lex` that support states, providing this information is trivial (although inefficient), as show in Figure 5. For other scanner generators, or if scanning efficiency is of great concern, other techniques can be used. The routine `src_line` stores a copy of the line using low-level string-handling support. While the routines used in these examples are tailored for `lex` semantics, the routines are generally wrapper routines for lower-level KHEPERA functions and would, therefore, be easy to implement for other front-end tools.

KHEPERA also handles interpretation of line number information generated by the C preprocessor. This requires a simple `lex` action:

```
^#\ .*        src_cpp_line(yytext, yyleng);
```

Finally, every scanner action must advance a pointer to the current position on the current line. This is accomplished by having every action make a call to `src_get(yyleng)`, a minor inconvenience that can be encapsulated in a macro.

The productions in the parser need only call KHEPERA tree-building routines—all other work can be reserved for later tree walking. This tends to simplify the parser description file, and allows the implementor to concentrate on parsing issues during this phase of development. A few example `yacc` productions are shown in Figure 6. The second argument to `tre_mk` is a pointer to the (optional) source position information obtained during scanning. The abstract representation of the constructed AST is that of an n-ary tree, and routines are available to walk the tree using this viewpoint.[1]

Immediately after the parsing phase, the AST is available for printing. Without any pretty-printer description, the AST is printed as a nested S-expression, as shown in Figure 7.

4.5 Pretty-printing

For pretty-printing, KHEPERA uses a modification of the algorithm presented by [9]. This algorithm

[1]Physically, the tree is stored as a rotated binary tree, although other underlying representations would also be possible.

```
NL              \n
...
%%
<INITIAL>{
   .*{NL}       src_line(yytext,yyleng); yyless(0); BEGIN(OTHER);
   .*           src_line(yytext,yyleng); yyless(0); BEGIN(OTHER);
}
...
{NL}            BEGIN(INITIAL);
```

Figure 5: Storing Lines While Scanning

```
Statement: Identifier '=' Expression
           { $$ = tre_mk(N_Assign, $2.src,
                         $1, $3, 0); }
         ;

StatementList: Statement
               {
               $$ = tre_mk(N_StatementList,
                           tre_src($1),
                           $1, 0);
               }
             | StatementList Statement
               {
               $$ = tre_append($1, $2);
               }
             ;
```

Figure 6: Building the AST While Parsing

is linear in space and time, and does not backtrack when printing. The implementation was straightforward, with modifications added to support source line tracking and formatted pretty-printing. Other algorithms for pretty printing, some of which support a finer-grain control over the formatting, are presented in [7, 2, 15, 16].

For each node type in the AST, a short description, using `printf`-like syntax, tells how to print that node and its children. If the node can have several different numbers of children, several descriptions may be present, one for each variation. List nodes may have an unknown number of children. Multiple descriptions may be present for multiple languages, with "fallback" from one language to another specified at printing time (so, Fortran may be printed for all of those nodes that have Fortran-specific descriptions, with initial fallback to unlabeled nodes (perhaps for C or for the original DSL), and with final fallback to generic S-expressions). This fallback scheme provides usable pretty-printing during development, even before the complete pretty-printer description is finished and debugged.

For printing which requires local analysis, implementor-defined functions can be used to return pre-formatted information or to force a line break. These functions are passed a pointer to the current node, so they have access to the complete AST from the locus being printed. While the pretty-printer is source-language independent and is unaware of the specific application-defined attributes present on the AST, the implementor-defined functions have access to all of this information. We typically use these functions to format type information or to add comments to the generated source codes.

Additional pretty-printer description syntax allows line breaks to be declared as "inconsistent" or "consistent"[2]; allows for forced line breaks; and permits indentation adjustment after breaks.

[2]See [9] for details. Each group may have several places where a break is possible. An inconsistent break will select one of those possible places to break the line, whereas a consistent break will select *all* of these places if a break is needed anywhere in the group. This allows the following formatting to be realized (assuming breaks are possible before +):

```
Inconsistent

( x = a + b + c
      + d + e + f)
Consistent

( x = a
      + b
      + c
      + d
      + e
      + f)
```

Original Program:

```
A = range(depth=0, 3)
B = (/ i in A : i + i /)
C = (/ i in A :
      (/ j in range(depth=0, i) :
         i /) /)
```

Initial AST (with attribute values shown after the slash):

```
(N_StatementList
  (N_Assign
    (N_Identifier/"A")
    (N_Call
      (N_Identifier/"range")
      (N_ExpressionList
        (N_Integer/3))))
  (N_Assign
    (N_Identifier/"B")
    (N_SequenceBuilder
      (N_Iterator
        (N_Identifier/"i")
        (N_Identifier/"A"))
      (N_Add
        (N_Identifier/"i")
        (N_Identifier/"i"))))
  (N_Assign
    (N_Identifier/"C")
    (N_SequenceBuilder
      (N_Iterator
        (N_Identifier/"i")
        (N_Identifier/"A"))
      (N_SequenceBuilder
        (N_Iterator
          (N_Identifier/"j")
          (N_Call
            (N_Identifier/"range")
            (N_ExpressionList
              (N_Identifier/"i"))))
        (N_Identifier/"i")))))
```

Figure 7: Example Input and Initial AST

4.6 The KHEPERA Transformation Language

KHEPERA transformations are specified in a special "little language" that is compiled into C code for tree-pattern matching and replacement. A simple transformation rule conditionally matches a tree, builds a new tree, and performs a replacement. The rule that implements the first sequence comprehension elimination transformation (Rule 1 from Section 4.3) is shown in Figure 8.

```
rule eliminate_iterator1
{
   match (N_SequenceBuilder
           (N_Iterator id1:N_Identifier D:.)
           id2:N_Identifier)
   when (tre_symbol(id1)
         == tre_symbol(id2))
   build new with D
   replace with new
}
```

Figure 8: Simple Transformation Rule

In Figure 8, a tree pattern follows the **match** keyword. Tree patterns are written as S-expressions for convenience. The tree pattern in this example is compiled to the pattern matching code shown in the first part of Figure 9 (code for sections of the rule follow the comment containing that section).

The **when** expression, which contains arbitrary C code, guards the match, preventing the rest of the rule from being executed unless the expression evaluates to true. The **build** statement creates a new subtree, taking care to copy subtrees from the matched tree, since those subtrees are likely to be deleted by a **replace** command.

The tracking necessary for debugging and transformation replay is performed at a low-level in the KHEPERA library. However, the KHEPERA language translator automatically adds functions (with names starting with `trk_`) to the generated rules. These functions add high-level descriptive information which allows fine-grain navigation during transformation reply, but which is not necessary for answering debugger queries.

A more complicated KHEPERA rule is shown in Figure 10. This rule implements the third sequence comprehension elimination transformation (Rule 3 from Section 4.3).

The example in Figure 10 uses the **children** statement to iterate over the children of the

```
int rule_eliminate_iterator1( int *_kh_flag, tre_Node _kh_node )
{
   const char *_kh_rule = "rule_eliminate_iterator1";
   Node _kh_pt;
   Node this = NULL; /* sym */
   Node id1 = NULL; /* sym */
   Node D = NULL; /* sym */
   Node id2 = NULL; /* sym */
   Node new = NULL;

   /* match (this:N_SequenceBuilder
             (N_Iterator id1:N_Identifier D:.) id2:N_Identifier) */

   _kh_pt = _kh_node;
   if (_kh_pt && tre_id( this = _kh_pt ) == N_SequenceBuilder) {
      _kh_pt = tre_child( _kh_pt ); /* N_Node */
      if (_kh_pt && tre_id( _kh_pt ) == N_Iterator) {
         _kh_pt = tre_child( _kh_pt ); /* N_Node */
         if (_kh_pt && tre_id( id1 = _kh_pt ) == N_Identifier) {
            _kh_pt = tre_right( _kh_pt );
            if (_kh_pt) {
               D = _kh_pt;
               _kh_pt = tre_parent( _kh_pt );
               _kh_pt = tre_right( _kh_pt );
               if (_kh_pt && tre_id( id2 = _kh_pt ) == N_Identifier) {
                  _kh_pt = tre_parent( _kh_pt );
                  assert( _kh_pt == _kh_node );

                  /* when (tre_symbol(id1) == tre_symbol(id2)) */

                  if (tre_symbol(id1) == tre_symbol(id2)) {
                     trk_application( _kh_rule, _kh_node );

                     /* build new with D */

                     new = tre_copy(D);

                     /* replace with new */

                     ++*_kh_flag;
                     trk_work( _kh_rule, _kh_node );
                     tre_replace( _kh_node, new );
                  }
               }
            }
         }
      }
   }
   return 0;
}
```

Figure 9: Generated Tree-Pattern Matching Code

```
rule dp_func_call
{
    match (this:N_SequenceBuilder
           iter:N_Iterator
           (f:N_Call
            fn:N_Identifier
            plist:N_ExpressionList))

    build newPlist with (N_ExpressionList)
    children plist {
        match (p:.)
        build next with (N_SequenceBuilder
                         iter p)
        do { tre_append(newPlist, next); }
    }

    build call with (N_Call fn newPlist)
    delete newPlist
    do { call->prime = f->prime + 1; }
    replace with call
}
```

Figure 10: Iterator Distributing Transformation Rule

N_ExpressionList node, and uses the **do** statement as a general-purpose escape to C. This escape mechanism is used to build up a new list with the tre_append function, and to modify an implementor-defined attribute (prime).

KHEPERA language features not discussed here include the use of a conditional **if-then-else** statement in place of a **when** statement, the ability to break out of a **children** loop, and the ability to perform tree traversals of matched subtree sections (this is useful when an expression must be examined to determine if it is independent of some variable under consideration).

4.7 Debugging with KHEPERA

The KHEPERA library tracks changes to the AST throughout the transformation process. The tracking is performed, automatically, at the lowest levels of AST manipulation: creation, destruction, copying, and replacement of individual nodes and subtrees. This tracking is transparent, assuming that the programmer always uses the KHEPERA AST-manipulation library, either via direct calls or via the KHEPERA transformation language, to perform all AST transformations. This assumption is reasonable because use of the KHEPERA library is required to maintain AST integrity through the transformation process. Since the programmer does not have to remember to add tracking capabilities to his transformations, the overhead of implementing debugging support in a DSL processor is greatly reduced.

The tracking algorithms associate the tree being transformed (T_i in Figure 1), the transformation rule (τ) being applied, and the specific changes made to the AST. This information can then be analyzed to answer queries about the transformation process. For example, the DSL implementor may have identified two intermediate ASTs, T_i and T_{i+1}, and may ask for a summary of the changes between these two ASTs.

On a more sophisticated level, the user may identify a node in the DSL program and request that a breakpoint be placed in the program output. An example of this is show in Figure 11. Here, the user clicked on the scalar + node in the left window. In the right window, the generated program, after 13 transformations have been applied, is displayed, showing that the breakpoint should be set on the call to the vector add function.

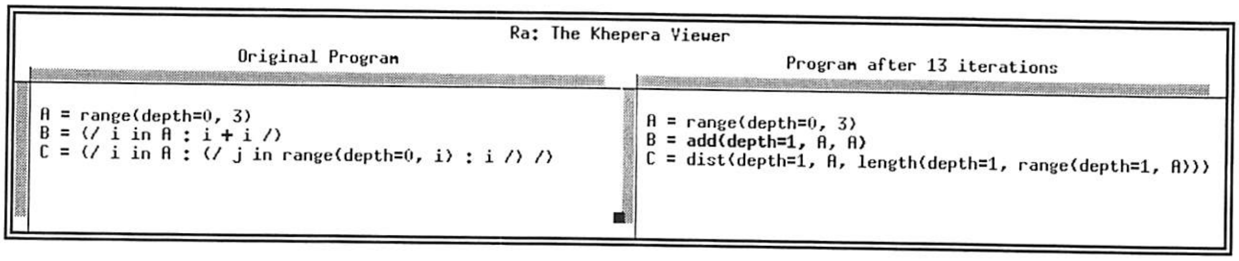
At this point, the user could navigate backward and forward among the transformations, viewing the particular intermediate ASTs which were involved in transforming the original + into the call to add. The ability to navigate among these views is unique to the KHEPERA system and helps the user to understand how the transformations changed the original program. This is especially useful when many transformations are composed.

The tracking algorithms can also be used to understand relationships between variables in the original and transformed programs. For example, in Figure 12, the user has selected an iterator variable i which was removed from the final transformed output. In this case, both occurrences of A are marked in the final output, showing that these vectors correspond, in some way, to the use of the scalar i in the original input.

In addition to the "forward" tracking, described here, KHEPERA also supports reverse tracking, which can be used to determine the current execution point in source terms, or to map a compile or run-time error back to the input source.

5 Conclusion and Future Work

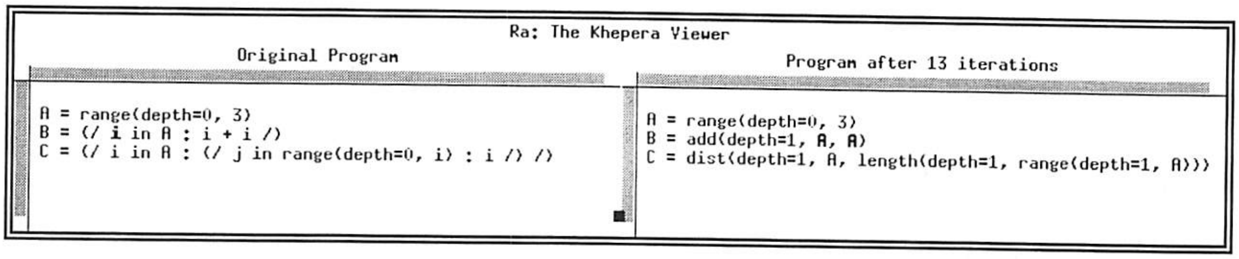
In this paper, we have presented an overview of our transformation-based approach to DSL processor implementation, with emphasis on how this approach provides increased ease of implementation and more flexibility during the DSL lifetime when compared with more traditional compiler implemen-



```
A = range(depth=0, 3)
B = (/ i in A : i + i /)
C = (/ i in A : (/ j in range(depth=0, i) : i /) /)
```

```
A = range(depth=0, 3)
B = add(depth=1, A, A)
C = dist(depth=1, A, length(depth=1, range(depth=1, A)))
```

Figure 11: Debugging with KHEPERA (Example 1)



```
A = range(depth=0, 3)
B = (/ i in A : i + i /)
C = (/ i in A : (/ j in range(depth=0, i) : i /) /)
```

```
A = range(depth=0, 3)
B = add(depth=1, A, A)
C = dist(depth=1, A, length(depth=1, range(depth=1, A)))
```

Figure 12: Debugging with KHEPERA (Example 2)

tation methods.

In the previous section we have provided an overview of the KHEPERA system using a small example. We have shown how the KHEPERA library supports AST construction and pretty-printing, and have demonstrated some of the capabilities of the KHEPERA transformation language and debugging system. Many additional features of the KHEPERA system are difficult to demonstrate in a short paper. These features include low-level support for common compiler-related data structures such as hash tables, skip lists, string pools, and symbol tables and for high-level functionality such as type inference and type checking. The availability of these commonly-used features in the KHEPERA library can shorten the time needed to implement a DSL processor.

Further, we have found that keeping lists of nodes, by type, can dramatically improve transformation speed. Instead of traversing the whole AST, we traverse only those node types which will yield a match for the current rule. However, since some transformations may assume a pre-order or post-order traversal of the AST, the "fast tree walk" problem is more difficult that simply keeping node lists: the lists must be ordered and the data structure holding the lists must be updateable during the tree traversal (this eliminates many balanced binary trees from consideration for the underlying data structure). We have found that an implementation based on skip lists [13] was viable—preliminary empirical results demonstrate a significant transformation speed compared with pattern matching over the whole AST. More details on this work will be presented in a future paper.

Another advantage of KHEPERA is the support for debugging via transformation replay. When the transformation are applied to the AST using the KHEPERA library support (with or without using the KHEPERA transformation language), then those transformations are tracked and can be replayed at a later time. KHEPERA includes support for arranging the transformations in an abstract hierarchy, thereby facilitating meaningful viewing by a DSL implementor. As part of a complete debugging system, KHEPERA also provides mappings which allow loci in the output source to be mapped back through the AST transformations to the input source (written in the DSL). These debugging capabilities are the subject of Faith's forthcoming dissertation.

6 Availability

Snapshots of the KHEPERA library, including working examples similar to those discussed in this paper, are available from `ftp://ftp.cs.unc.edu/-pub/projects/proteus/src/`.

References

[1] Jon L. Bentley, Lynn W. Jelinski, and Brian W.

Kernighan. CHEM—a program for phototypesetting chemical structure diagrams. *Computers and Chemistry*, **11**(4):281–97, 1987.

[2] Robert D. Cameron. An abstract pretty printer. *IEEE Softw.*, **5**(6):61–7, Nov. 1988.

[3] James R. Cordy and Ian H. Carmichael. *The TXL programming language syntax and informal semantics, version 7.* Software Technology Laboratory, Department of Computing and Information Science, Queen's University at Kingston, June 1993.

[4] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: a rapid prototyping system for programming language dialects. *Comp. Lang.*, **16**(1):97–107, Jan. 1991.

[5] Matt Englehart and Mike Jackson. ControlH: A Fourth Generation Language for Real-time GN&C Applications. *Symp. on Computer-Aided Control System Design* (Tucson, Arizona, Mar. 1994), Mar. 1994.

[6] J. Grosch and H. Emmelmann. *A tool box for compiler construction*, Compiler Generation Report No. 20. GMD Forschungsstelle an der Universität Karlsruhe, 21 Jan. 1990.

[7] Matti O. Jokinen. A language-independent prettyprinter. *Softw.—Practice and Experience*, **19**(9):839–56, Sep. 1989.

[8] Brian W. Kernighan. PIC—a language for typesetting graphics. *Softw.—Practice and Experience*, **12**:1–21, 1982. Published as AT&T Bell Laboratories (Murray Hill, New Jersey) Computing Science Technical Report No. 116: *PIC—a graphics language for typesetting (user manual)*, available as `http://cm.bell-labs.com/cm/cs/cstr/-116.ps.gz`.

[9] Derek C. Oppen. Prettyprinting. *ACM Trans. on Prog. Lang. and Sys.*, **2**(4):465–83, Oct. 1980.

[10] Daniel William Palmer. *Efficient execution of nested data-parallel programs.* PhD thesis, published as Technical report TR97-015. University of North Carolina at Chapel Hill, 1996.

[11] Terence John Parr. *Language Translation Using PCCTS and C++: A Reference Guide.* San Jose: Automata Publishing Co., 1997.

[12] Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. *Proc. 4th Annual Symp. on Princ. and Practice of Parallel Prog.* (San Diego, CA, 19–22 May 1993). Published as *SIGPLAN Notices*, **28**(7):119–28. ACM, July 1993.

[13] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, **33**(6):668–76, June 1990.

[14] Reasoning Systems. REFINE *user's guide*, 25 May 1990.

[15] Lisa F. Rubin. Syntax-directed pretty printing—a first step towards a syntax-directed editor. *IEEE Trans. on Softw. Eng.*, **SE-9**(2):119–27, Mar. 1983.

[16] Martin Ruckert. Conservative pretty printing. *SIGPLAN Notices*, **32**(2):39–44, Feb. 1997.

[17] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating Scalar Optimization and Parallelization. *Languages and Compilers for Parallel Computing (Fourth International Workshop)* (Santa Clara, California, 7–9 Aug. 1991). Published as U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science*, **589**:137–51. Springer-Verlag, 1992. An overview of a more recent version of SUIF is available as Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy, *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*, available at `http://suif.stanford.edu/suif/-suif-overview/suif.html`.

[18] Arie van Deursen and Paul Klint. Little languages: little maintenance? *Proceedings of DSL '97 (First ACM SIGPLAN Workshop on Domain-Specific Languages)* (Paris, France, 18 Jan. 1997). Published as *University of Illinois Computer Science Report*, `http://www-sal.cs.uiuc.edu/-~kamin/dsl/`:109–27, Jan. 1997.

DiSTiL: a Transformation Library for Data Structures

Yannis Smaragdakis and Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{smaragd,dsb}@cs.utexas.edu

Abstract

DiSTiL is a software generator that implements a declarative domain-specific language (DSL) for container data structures. DiSTiL is a representative of a new approach to domain-specific language implementation. Instead of being the usual one-of-a-kind stand-alone compiler, DiSTiL is an extension library for the *Intentional Programming* (IP) transformation system (currently under development by Microsoft Research). DiSTiL relies on several reusable, general-purpose infrastructure tools offered by IP that substantially simplify DSL implementation.

1 Introduction

In the past few years, the popularity of domain-specific languages has steadily increased. Such languages offer concise ways of expressing complex, domain-specific concepts and applications, which in turn can offer substantially reduced maintenance costs, more evolvable software, and significant increases in software productivity [Kie96, Bat97b, Due97, Die97]. Our research is in the design and implementation of software generators. Generators are compilers for domain-specific languages. Our particular research emphasis, which largely has distinguished our work from others in the generator community, is on generators that synthesize implementations of declarative specifications of domain-specific constructs through component composition. Thus, an integral part of our methodology, called *GenVoca* [Bat92], is to identify the fundamental building blocks of software construction for a target domain. GenVoca components actually define sophisticated program transformations that convert domain-specific language constructs into their host language implementations. In this way, a domain-specific program is reduced (transformed) into an executable host language program by a series of transformations, where each transformation corresponds to a component in a domain-specific transform library. The advantage of this approach is *scalability*: a small number of GenVoca components can be composed in vast numbers of ways to yield huge families of distinct implementations for domain-specific constructs [Bat93].

From our experience, only 30% of the effort in building GenVoca generators is actually spent on coding components (i.e., writing program transformations). The majority of the time is spent on infrastructure development (i.e., developing tools for representing programs as data, writing and composing components, validating component compositions, etc.). This overhead has substantially hindered the development of GenVoca generators under realistic time and funding constraints. Tools are needed both to reduce the effort in building generators and to promulgate their use.

In this paper, we present DiSTiL — a software generator for the domain of container data structures. The language of DiSTiL extends the C programming language with domain-specific constructs for specifying complex data structures declaratively. When a DiSTiL program is "compiled", the declarative data structure specifications are replaced by their C implementation, which is specified by a composition of DiSTiL components. In the following, when no confusion can arise, we will use the name DiSTiL to also mean the domain-specific language that the generator implements.

The overall design of DiSTiL is similar to that of the previously built GenVoca generators P1 and P2 [Sir93, Bat92, Bat97b]. However, its implementation is radically different. Instead of being a one-of-a-kind generator that is totally specific to the data-structure domain, DiSTiL is implemented as a *transformation library* for the *Intentional Programming (IP)* system [Sim95], which is currently under development by Microsoft Research. IP provides a domain-independent implementation substrate and tools that have substantially simplified the implementation of DiSTiL.

The novelty of DiSTiL is that it is the first truly complicated domain-specific generator that was built using IP. We found that IP, by itself, lacked certain features that were required to simplify DiSTiL's development. In this paper, we describe IP, our general-purpose extensions to it (called *generation scoping* — a general-purpose hygienic code generation facility [Sma96]), and DiSTiL — the language and its implementation. We argue that IP's infrastructure is well-suited for building compilers for DSLs, and that it substantially reduces the effort needed for their construction.

2 Microsoft's Intentional Programming (IP) System

Many domain-specific languages can be implemented as domain-specific extensions to existing programming languages. Up to now this approach had been examined mostly in the context of functional languages. LISP [Ste90] and its variants (e.g., Scheme [Cli91b]) have powerful language extension mechanisms (under the rather misleading name "macros") that are well-suited for DSLs. Unfortunately, using LISP as a software generator infrastructure has undesirable consequences for many applications: LISP's powerful meta-programming system is trapped inside a hard to optimize functional language. The syntax is strange, and many operations impose an unnecessary performance overhead to the unsuspecting user. Furthermore, every program has to pay the cost of garbage collection.

Nevertheless, there is no fundamental limitation preventing the application of the extension approach to other programming languages. The essential elements are a language extensibility mechanism and a powerful meta-programming system (i.e., constructs for representing programs as data). We note that the issue of extending imperative languages has been addressed before (e.g., [Wei93]). Microsoft's IP, however, is the first integrated programming environment specifically designed with language independence and language extensibility in mind. The next two sections describe the Intentional Programming infrastructure and the machinery used to build DiSTiL as a language extension.

2.1 The Intentional Programming Environment

IP [Sim95] is a language-independent programming environment. Language independence is achieved in IP by representing all source code (in whatever language) as an *abstract syntax tree (AST)*. Nodes of an AST are called *intentions* and correspond to semantic constructs of a language. Examples of intentions include if-statements, for-loops, type declarations, assignment-statements, etc. Thus, libraries of intentions can be created for representing programs in various programming languages. Many intentions are themselves language-independent; i.e., their semantic meaning (but not their syntax) is shared in many languages [Vil97]. The if-statement, for example, with a general form of an `if` operator and a 3-tuple argument `<boolean-expression, then-statement, else-statement>`, is a standard "intention" in virtually all programming languages.

The syntax (or external representation) of an intention is user-controlled. (For example, the syntax of an if-statement in C is different than in Pascal). This variability is captured by *unparsing* methods that are associated with intentions. Unparsing is the process of displaying an AST to the user for direct manipulation. In IP, unparsing is more than just pretty-printing — it is two-dimensional and fully graphical. That is, an intention may be represented as a complex image and can be positioned accordingly. This offers the possibility of developing special, non-ASCII notation for domain-specific languages (e.g., mathematical symbols). For instance, it is relatively straightforward to make the combinatorial intention `choose(n,m)` to unparse as $\binom{n}{m}$, in the usual mathematical notation.

The extensibility of IP lies in its ability to define new intentions and to define enzymes. New intentions express domain-specific programming constructs. In effect, adding intentions is equivalent to extending the grammar of the host language. *Enzymes* are transformations on ASTs, i.e., functions that replace an AST with another AST. Using enzymes, new intentions can be transformed into existing ones, effectively extending the language. The programming interface for transformations is procedural — pattern-based extensions are also provided as a higher-level concept. Although the interface for transformations is not yet final, it includes operations to traverse and create ASTs as well as operations to manipulate semantic information (for instance, variable and expression types). The semantic information elevates the interface above the usual syntactic macros (as, for instance, in LISP). A distinguishing factor between IP and existing transformation systems (e.g., TXL [Cor91], Refine/Dialect [Rea86, Rea89]) is the power of the transformation engine itself. The goal of IP is to have complete, industrial-strength languages implemented entirely as collections of enzymes. A complex transformation infrastructure is in place to accommodate this requirement. For example, sophisticated scheduling of transformations according to their dependencies is performed. Thus, transformations that were designed independently can be applied in such a way that they will not interfere with each other. Additionally, transformations may have non-local effects following a strictly defined protocol of permissions and information passing.

IP uses parsers for importing programs already written in conventional programming languages. This conversion is one-way, however. After a program is expressed as an AST it can be edited directly. IP provides a powerful structure editor for this purpose. Users edit unparsed versions of a program, but all text-like editing commands directly manipulate the underlying AST. This enables enzymes to be applied at editing time. For instance, it is possible to use a standard syntactic rewrite like a DeMorgan transform of boolean expressions both as an editing enzyme and as a compilation enzyme. A user can select/highlight a boolean expression during editing and invoke the DeMorgan enzyme (for instance, to turn an OR expression into an AND expression for readability). The same enzyme could be automatically applied by IP during compilation (for instance, as an optimization).

The IP environment is fully configurable. Opening a source code document that refers to domain-specific intentions can cause new commands to be added in the environment window menus, new buttons and toolbars to appear, etc. In this way, the author of a domain-specific language implementation can also customize the development environment for language users.

It should be clear from the above that IP is an appropriate platform for implementing domain-specific languages. It allowed us to concentrate on the task of devising powerful data structure abstractions without worrying about infrastructure support. At the same time, DiSTiL is an example of the applications that IP was intended to support. It is a powerful domain-specific language that can be transparently integrated into the system as an extension and take full advantage of its transformational capabilities.

2.2 Generation Scoping

As a transformation library, DiSTiL deals extensively with manipulating code fragments. To facilitate our work, we designed the *generation scoping* mechanism: a meta-programming system for IP in which DiSTiL components are expressed. The system consists of *code template operators*, similar to the `backquote` and `comma` operators of the LISP language. Generation scoping is a general-purpose facility oriented towards large-scale code generation and was not designed to support only DiSTiL. This section reviews its essential features and applicability. A more complete discussion is in [Sma96][1].

Meta-programming systems are notorious for introducing ambiguities regarding the environment in which generated variable references are resolved. Programming languages usually determine the meaning of identifiers using their position in a program. Generated programs, however, are usually composed from small fragments. In this case we are usually unaware of the final position/scope of a fragment in the generated code. Thus, it becomes a bad practice to let the position of identifiers in the final program determine their meaning — erroneous references can easily be introduced.

This problem has been studied extensively in the context of macro expansion and systems that address it are called *hygienic* (e.g., [Koh86], [Cli91a], [Wal97]). A complete solution comes in the form of hygienic, lexically-scoped macros (see [Cli91a]). As we explain in [Sma96] the standard macro-expansion methods are not directly applicable to software generators. Instead we had to develop the generation scoping system which is in many ways similar to the lexically scoped macros machinery of [Cli91a] but is better suited for generator development. The difference is that the environments which determine identifier bindings become first-class objects and can be manipulated directly. Most importantly, environments can be organized hierarchically into directed graphs with every environment having access to all others reachable in the graph. This adds significant power: instead of a hygienic mechanism for small, self-contained units (macros) we get a method that can handle complex scoping in the generated program, independently of the target language [Sma96].

Generation scoping includes standard operators to designate code templates and escapes from them: `quote`, written `'`, and `unquote`, written `$`. Also it allows explicitly closing a code fragment when it is generated in an environment where identifiers have specific meanings. This is done using the `environment` operator around one or more code templates (quoted code fragments). For instance, the code fragment:

```
environment(E)
  Output('int i = 0;);
```

will generate code that declares variable `i` as an integer and initializes it. The code is generated in environment `E` (assumed to have been declared before). The system can detect that `i` is being declared in this fragment.

1 Generation scoping is actually yet another example of an embedded domain-specific language. In this case, it is a language to express and compose lexically-scoped code fragments.

Operator	Description
`'<code_template>`	***Quote operator.*** Generates a code fragment according to `code_template`. Similar to LISP `backquote`.
`$<code>`	***Escape operator.*** Executes `code` inside a quoted fragment.
`environment` `(<Env_id>) <code>`	Evaluates quoted code in a given "environment". Variable references in quoted code will be resolved relative to this environment.
`SetParent` `(<Env_id1>,<Env_id2>)`	Organizes environments hierarchically. Quoted code in the child environment will be able to view variables in parent environment as well.
`alias` `(<tree_expr>, <variable>)`	Sets the value of identifier `variable` to `tree_expr`. Every time `variable` appears in quoted code (in the same environment as the **`alias`** command) it will be replaced by `tree_expr`.

Figure 1: Generation Scoping operators

Therefore, `i` now becomes a declared variable in environment `E` and future occurrences of identifier `i` in the same environment will refer to the variable declared above. By explicitly choosing the environment (scope) of a generated code fragment we can completely dissociate variable scoping from the variable's position in the generated program. This ability is used in DiSTiL to ensure that identifiers are bound to the correct variables.

Other important generation scoping operators include `SetParent` and `alias`. `SetParent` is used to organize environments hierarchically, in much the same way lexical scopes are hierarchically organized by nesting. That is, a "child" environment has access to variables introduced in a "parent" environment but variables with the same name in the "child" eclipse variables in the "parent". `alias` is used to introduce symbolic names for complex generated expressions. A table of the main generation scoping operators is shown in Figure 1. Examples of the use of generation scoping can be found in [Sma96].

3 DiSTiL

3.1 Motivation

A central problem in software development is the creation, maintenance, and evolution of data structures. Initially, with a partial understanding of the system requirements, a programmer invents data/storage structures to address a perceived need. These data structures are then either implemented by hand (a tedious process) or taken from a component library (e.g., STL [Ste95], or the Booch components [Boo87]) . It is quite rare, however, that the projected requirements are accurately reflected in the first design (and even if they are they may change in time). Altering a data structure is often costly; interfaces to different data structures can vary widely, and thus may require extensive source code modifications, leading to yet another (expensive) round of coding and debugging.

We believe that data structures should not have ad hoc interfaces. Instead they should provide a stable, well-designed interface that insulates applications from changes to data structure implementations. This, incidentally, is also the premise behind the C++ Standard Template Library — STL [Ste95]. STL, however, does not take this idea to its logical conclusion. Compatibility is limited to a specific level, while different kinds of STL data structures (e.g., sequences and associative containers) still have different interfaces. This significantly restricts the interchangeability of data structures. Moreover, STL only offers rather elementary data structures. Complex data structures must be implemented by hand. For example, if elements of a container are to be simultaneously linked onto two key-ordered lists, or a key-ordered list (for sequential accessing) and a hash-table (for fast key accessing), STL users have to either (a) write their own, customized STL component to accomplish the task or (b) devise ways of integrating existing STL components manually, and write source code that maintains the correctness of these structures when element keys are updated. Both approaches are unpleasant and preclude the ease of evolving data structure implementations.

We believe a different approach is needed — one based on a declarative language that is specific to the domain of data structures, rather than using typical component libraries (link libraries, macro/template libraries, binary components, etc.). Our language, called DiSTiL, extends the C programming language with declarative

goto_first:	Set cursor to first legal position
goto_next/goto_prev:	move cursor forward/back
goto_nth	: move cursor to n-th ordered position
is_legal	: is the cursor in a legal position?
foreach	: iterate over all elements in cursor range
insert/delete:	insert/delete current element
getrec/ref:	return current record or single field
update	: change value of field in current record

DiSTiL cursor operations

open_cont	: open/initialize the container
close_cont:	close the container
size	: return the total number of elements
is_full	: is the container full?

DiSTiL container operations

Figure 2: Set of DiSTiL operations

statements/operations on data structures. These statements isolate the actual data structure implementation from the application itself, thereby allowing radically different implementations of data structures to be evaluated without requiring modifications to the application's source code [Bat93-95a]. It is the responsibility of the compiler to ensure efficiency. As an added benefit, the ability to reason about programs is greatly enhanced, often allowing for automatic design checking mechanisms and high-level optimizations [Bat97a].

3.2 The DiSTiL Programming Language

All data structures in DiSTiL are modeled using *containers* and *cursors*. Essentially, we view all data structures in our universe as pairs of containers and cursors (iterators). These two facets explicitly decouple the notion of element storage from that of element access. The cursor-container pair provides the only interface the user has to a data-structure. When viewing a data structure as a collection of elements, the most important operation that can be defined is that of a *selection*. A selection gives the user a way to define a subset of a collection according to a certain *selection criterion*. In the case of DiSTiL, the effect is achieved by assigning *selection predicates* to cursors. Such predicates may express an arbitrary relation on the values of the fields of stored elements. A cursor is guaranteed to only access elements satisfying its selection predicate. Additionally the user may specify the order of retrieval as an ordering relation on element fields. *The mechanisms of order and predicate specification are the only way for the user to control element access.* The system is otherwise free to implement data structure operations in any semantically correct way.

An abbreviated example of container-cursor specifications in DiSTiL is given below.

```
typedef struct {
   char[8] phone;
   char[31] name;
} phonebook_record;
 // C struct declaration

Container (phonebook_record) cont1;
 // abbreviated container declaration

Cursor (cont1, phone == "4783487")
   curs1;
 // cursor declaration
Cursor (cont1,
        name > "Sm" && name < "Sn",
        ascending(name)) curs2;
 // another cursor declaration
```

This example presents a phone-book data structure. We begin by declaring the record type for the elements as a C type. Then a container of elements and two different cursors on that container are declared. The first ranges over all elements (probably a single one) with phone number "4783487". The second selects all records in the data structure with a name that begins with "Sm", in ascending order.

DiSTiL offers a standard set of operations on containers and cursors, regardless of the actual data structure implementation. The code fragment below illustrates the `foreach` construct, which is used to iterate over elements selected by a cursor. The element in the current cursor position can be examined, updated, or deleted using standard cursor operations (a summary of all DiSTiL operations appears in Figure 2).

```
foreach (curs1) {
       // for each selected entry
   printf("%s", ref(curs1, name));
       // print name
   update(curs1, phone, "4718731");
       // change phone number
}
```

The interface to DiSTiL data structures does not depend on the actual data structure used. This way DiSTiL programs can stay the same for different data structure implementations. For example, the phone-book could be implemented as an ordered linked list, a binary-tree, a hash-table, or any other structure or combinations of structures. Nevertheless, the above program fragment would remain the same across all different implementations.

It is worth noting that DiSTiL cursors and containers can be composed arbitrarily. Thus we can have a data structure storing cursors, or containers, or containers of containers, etc. This can yield interesting data structure configurations in their own right (i.e., we can explicitly create complex indexes to data structures using containers of cursors).

In effect, we are giving a relational front-end to container data structures [Bat93]. Using relational abstractions to hide data structure details is not new (e.g., see [Coh89, Coh93]), but our ability to couple relational abstractions with component technologies to generate vast families of efficient implementations is novel. In fact, readers might note that our work parallels recent advances in *object-oriented databases (OODBs)* to make them more extensible. Extensible DBMS technologies were developed in the mid-1980s, and one of the original projects was Genesis. Genesis was the first (our first) GenVoca generator [Bat88a-b]: it was also the first technology for assembling relational database systems from components.[2] Combinations of Genesis components produced different relational DBMSs with vastly different implementations. Our work on P3 [Bat97b], and now DiSTiL, can be viewed as a continuing evolution of the Genesis work. It exposes the relationships between domain-specific languages, component-based generators, and the synthesis of high-performance domain-specific software. We could add many more features that would put DiSTiL on par with the embedded capabilities of object-oriented databases, but this has not been our research emphasis or interest.

3.3 Implementation Specification Using Component Compositions

DiSTiL applications can define the features of their data structures and declare how their implementations are to be generated. At the top of a DiSTiL program is a specification of how DiSTiL constructs are to be implemented. This specification, called a *type equation*, is a named composition of DiSTiL components. Each DiSTiL component implements a sophisticated program transformation that encapsulates a primitive building block of container data structures.

As an example, we will show how the phone-book of Section 3.2 can be implemented as a hash table (DiSTiL `Hash` component) in conjunction with a red-black tree (`Tree`) with elements that are allocated when needed (`Malloc`) from main memory (`Transient`). The corresponding type equation appears below. To alter or evolve the data structure merely requires altering the container's type equation and re-compiling; no other source code modifications are needed.

```
typeq (phonebook_record,
       Hash(Tree(Malloc(Transient))))
   typeq1;
 // type equation specification
```

The actual container that will hold the elements is declared below. At container declaration time it is specified that the hash table is organized by phone number (for fast lookups by phone) while the red-black tree has the name field as its key (for fast retrievals of alphabetically ordered names). In database terminology, we are organizing our data with a red-black tree index on the name field and a hash table index on the phone.

```
Container (typeq1,
           (Hash(phone), Tree(name)))
   cont1;
 // container declaration
```

Using component compositions to express complex entities (see [Nei80]) is a hallmark of the scalability of GenVoca. Customized software systems implement *m* features out of a possible *n* features. Rather than building an exponential number of monolithic systems that offer unique sets of features, one should build systems by composing primitive components that encapsulate individual features. Thus, by making feature combinatorics explicit, it is possible to describe vast families of systems with a relatively small number of components. The set of components that implement the same interface is called a *realm*. A realm is, in effect, a library of plug-compatible and interchangeable components. A summary of DiSTiL realms and components can be found in Appendix A.

2 A similar approach to relational database system extensibility was later (and independently) developed and deployed in IBM's Starburst project [Haa90].

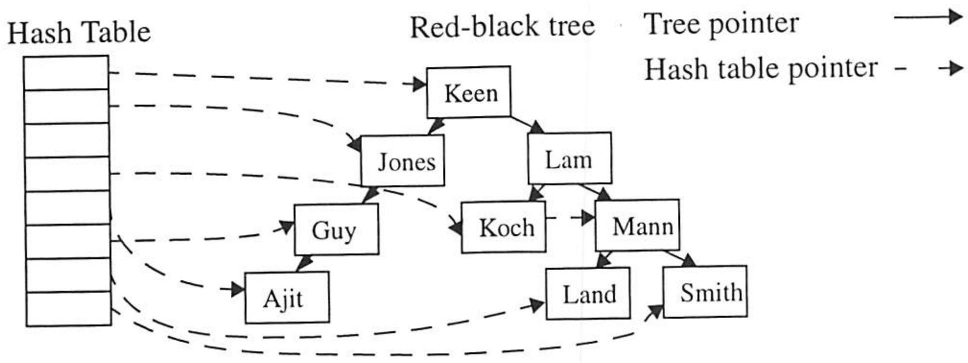


Figure 3: Phone book example

3.4 The DiSTiL Generator

For a given type expression, it is the responsibility of the DiSTiL generator to apply the transformations prescribed by each of the participating components. The generator will thus replace all DiSTiL operations by their corresponding C implementation. The resulting C program can then be compiled and executed. As a higher level language, DiSTiL offers significant leverage to its compiler, allowing it to perform powerful optimizations and error checking.

We illustrate DiSTiL's actions with a concrete example. The cursor definitions of Section 3.2 are replicated below for quick reference.

```
Cursor (cont1, phone == "4783487")
    curs1;
Cursor (cont1,
        name > "Sm" && name < "Sn",
        ascending(name))
    curs2;
```

Consider any of the DiSTiL cursor operations on `curs1` when the container uses the type expression of Section 3.3 (hash table and red-black tree indexes). The most efficient way to retrieve elements satisfying the predicate on `curs1` is to use the hash index on the phone number. DiSTiL can *statically* determine this by analyzing the predicate, estimating the cost of the retrieval using each available index and selecting the data structure that offers the lowest cost. For example, a `goto_first` DiSTiL operation[3] on `curs1` is implemented using code that hashes the given phone number and follows the hash table links until it finds the first element satisfying the desired predicate. Similarly, operations on `curs2` should be transformed only in terms of the red-black tree structure, since this is the most straightforward way to locate records lexicographically by name. If the composition describing our data structure changes (for instance, if we decide we want a second red-black tree in place of the hash table), DiSTiL will re-evaluate the cursor predicate and choose an appropriate way to implement its operations in the new layout without requiring any programmer intervention.

Figure 3 shows a possible run-time configuration of the data structure. Keeping it as a guide, let's see what happens if we change the phone number in one of our records (for instance, "Keen"). The change is one that would regularly be encountered in a realistic setting — people's phone numbers change. The most straightforward way to update the structure is to remove the affected element and re-insert it after the change has been performed. DiSTiL, however, can do better than this: Since only the phone number changes, the record can stay in its original place in the tree. The only structure that needs to be updated is the hash table, which is organized by phone number. DiSTiL detects this by analyzing the `update` operation according to the field being updated. Subsequently, the operation is transformed into code that performs the corresponding changes only to the primitive structures that actually depend on the changed field. In our case, the "Keen" record will remain at the root of the tree but its phone number will be re-hashed in the table possibly making the record to be linked in a different place. Again, the mechanism is straightforward. All DiSTiL components that support the `update` operation implement it according to the pseudo-code of Figure 4. The element is removed and re-inserted only if the update actually

3 The first six cursor operations of Figure 2 are retrieval operations (that is, they are implemented using the most efficient data structure available) while the last five will propagate through all components in a type expression.

```
CODE update(CODE field, CODE new_value)
{
   if (field == key_field)
      return '(
         {  $unlink();
            $update_code(field, new_value);
            $link();  });
   else return update_code(field, new_value));
}
```

Figure 4: Update operation (implementation dependent)

affects the key for this particular component. This optimization is a typical example of *partial evaluation*: the operation is specialized at compile time by exploiting a restriction on its (implicit or explicit) parameters. This is just one of the many cases where partial evaluation (in the form of specialization) is used in DiSTiL.

Additionally, the DiSTiL specification offers itself for easy checking of design consistency. A composition and the operations performed on it can be validated to ensure that they are meaningful. This is the role of the *design-rule checker* of DiSTiL. DiSTiL components come with higher-level knowledge about their properties (explicitly encoded using boolean attributes). This information is used to express domain-specific properties like "this component does not leave the cursor in a valid position after deletion" or "this component keeps track of the data structure size". Compositions are checked in two ways: The system ensures that all their components can co-exist and that they support all operations performed on the particular composition. In other words, compositions may be correct relative to a certain set of operations but not to another (for instance, a certain data structure may not support deletions). It is of interest to examine the results of the design rule checker application. The attributes associated with components are at a much higher level than regular static information (types) in programming languages. As a consequence, the checking mechanism can provide more informative, comprehensive and accurate error messages [Bat97a].

The previous examples are indicative of the flexibility afforded by DiSTiL through use of meta-level reasoning about the code it produces. DiSTiL components are "intelligent": They exchange information to coordinate their actions. This interaction can make code easy to evolve (DiSTiL handles the changes automatically) even though the actual model of operation (for instance, the structures actually used to implement a retrieval) has changed. The ability to process predicates and reason about the best way to implement the associated operations is what sets DiSTiL apart from usual data structure libraries. In short, cursor actions in DiSTiL are specified *intentionally* (declaratively) instead of operationally. The system transforms specifications into efficient code using its knowledge of the characteristics of the given data structure. Such knowledge may include the number and kind of indexes on the stored elements and information specific to each of the data structure components (for instance, that a binary tree is an ordered structure). Additionally, the domain-specific knowledge encoded in DiSTiL components enables higher-level error checking of component compositions. With DiSTiL the level of programming is effectively raised, resulting in code that is much easier to understand and that can straightforwardly evolve to match changing needs.

3.5 Generator Implementation

The DiSTiL generator is a 10Kloc (thousands of lines of code) library that operates on top of the Intentional Programming system. The way to interface with user programs is through the use of new intentions (like the `container` and `cursor` types and the `insert`, `goto_first`, etc. operations). In other words, DiSTiL keywords are introduced as linguistic extensions to IP. Each DiSTiL component implements all the interface operations of Figure 2. The implementations are integrated in a top-down fashion to produce the final transformation for the given operation.

Consider again the example of Section 3.3 (structure with hash table and red-black tree indexes). When an `insert` operation is transformed, the hash table component contributes code of the form:

```
'( if (Container->bucket[i] == 0)
     {  Container->bucket[i] =
               Cursor.obj;
        Cursor.obj.next = NULL;   }
   else ... )
```

Similarly, all other components contribute their own insertion code. For instance, the red-black tree component creates the insertion code fragment

```
'( ELE_TYPE* y = &Container->header;
ELE_TYPE* x =
      Container->header.parent;
while (x != Container->NIL)
    ...
... )
```

The hash table component (being first in the composition order) determines how the two code fragments are composed. In this case the composition is as simple as adding the hash table code right after the red-black tree insertion code.

The above code expressions are generated in distinct generation environments (see Section 2.2). This way, for instance, the expression `Container->bucket` is only legal in the context of hash table operations. Isolating the generation environment for each instance of a component allows us to specify component code in an abstract form. If a certain composition contained three different containers, each of which is organized using two different red-black trees, the expression `Container->header` would still be unambiguous. All the context information is captured in the generation environment structure (which only has to be set up once). The interested reader is referred to [Sma96] for a more complete example.

In the IP framework no parser is needed for the DiSTiL language — instead we have added support for editing source code with DiSTiL operators directly (in abstract syntax tree format using the IP graphical structure editor). The new primitives are given the right properties to be correctly displayed on screen. The unparsing (display) in this case is straightforward (for instance, we have to make sure that the `foreach` primitive is displayed as an iterator with its last argument being a statement instead of an expression). The graphical unparsing capabilities of IP are used in several other cases as well. For example, code templates (quoted code fragments) in DiSTiL components can be displayed in special styles or colors, etc.

The result of compiling DiSTiL in the IP system is a DLL (dynamic link library) with system extensions. To create a DiSTiL program, the user includes a DiSTiL interface file as a library. This causes the DLL to be loaded. The new intentions can now freely be used in an IP source file — the file becomes a DiSTiL specification. Every time such a file is transformed (compiled) the compiler will dispatch to the DLL to handle DiSTiL-specific intentions. DiSTiL will validate the compositions used, type-check the arguments, and possibly report error-messages using standard IP interfaces. If no errors are detected, it will compose abstract syntax tree fragments to create an implementation for the given operation expressed using only core IP primitives. The result is a program generated through transformation and (possibly) linking to a static library. The static library implements component aspects orthogonal to DiSTiL (like hash functions and persistent storage routines).

The efficiency of the compilation process is obviously a major concern in any transformation system. Since IP is still under development, it is too early to judge the speed of compilation for code with transformation extensions. In our experience, however, transformation from DiSTiL primitives to C code accounted for only a small part of the time spent in compiling.

Detecting specification errors is the responsibility of the DiSTiL design-rule checker. Each DiSTiL component has two logical propositions and two boolean functions associated with it. The first two express the conditions that the component expects the layers "above" it and "below" it to satisfy. The notions of "above" and "below" refer to the ordering of components in a composition (see Section 3.3). The two boolean functions specify the properties of the component in terms of the properties of the layers above and below it. Using these logical attributes the correctness of a composition can be checked with a single traversal of the composition structure. Additionally, DiSTiL operations can impose a condition on the properties of the entire composition. This can ensure that a certain operation is supported in the given type expressions. The mechanics of this validation are rather straightforward and the interested reader can find more details in [Bat97a].

4 Lessons Learned

The design and implementation of DiSTiL gave rise to several interesting observations of wider applicability in the area of domain-specific languages. Below, we discuss some of the lessons we have learned.

Domain-Specific Language Design. Abstracting away the details of an implementation leads to simpler and more powerful specifications. This is the principle behind higher-level languages and it is exploited fully in DiSTiL. So the main goal of a DSL design should be a significant rise in the level of abstraction that the user is exposed to. Thus, one can radically alter the implementation of a specification without needing to modify the application source. Ideally we would like to have implementation-independent declarative specifications. Designating *what* needs to be done and not *how* it should be done is the first step in the abstraction process. No DSL, however, is usable unless it offers acceptable performance. Otherwise an abstract specification can only serve as a design guideline and the actual implementation will have to result from manual refinement of the specification. The challenge for the DSL designer is to discover a level of abstraction that is amenable to automated reasoning for error-checking and optimization. In this way, the advantages of both abstraction and efficiency can be obtained.

These observations are clear in DiSTiL. All DiSTiL data structure components have a common interface and operations on them are specified through a mix of operational and declarative primitives. The DiSTiL generator can automatically perform the right operation to the right data structure by analyzing the predicate associated with the current cursor. Thus, the two main benefits of domain-specific languages are obtained: The design goal of simplicity is satisfied by the abstract interface and the practical goal of efficiency is satisfied by the generator optimizations. Many of the same optimizations have been applied in the past in the P2 lightweight DBMS generator with impressive results [Bat97b]. It should be noted that the optimizations are applied on top of quite efficient individual component implementations. Our red-black tree component, for instance, without any optimizations, will produce code almost identical to the most common implementation of the STL tree component (the, publicly available, Hewlett Packard implementation of STL).

Domain-Specific Language Implementation. In writing DiSTiL, we found it valuable to have an extensible system (IP, in our case) in which a DSL can be expressed as incremental changes. IP relieves the generator writer from having to parse source code constructs and to maintain source information (for instance, scope). At the same time it offers an easy mechanism for specifying transformations, reporting errors to the user, etc. Furthermore, the extensibility of the system can be exploited during generator implementation. Any machinery commonly used in generators can be expressed as an extension and reused exactly as if it was a part of the original language. In other words, during the implementation of DiSTiL we created domain-specific extensions for the domain of generators itself (e.g., generation scoping of Section 2.2). Generation scoping is a powerful hygienic meta-programming system — a valuable layer of infrastructure for all generators transforming their primitives into code in a high-level language.

To realize the flexibility afforded by the DiSTiL design, consider the P2 generator. P2 used its own parser, data-definition language, component specification language and back-end [Bat97b]. All these elements were specific to data-structures and changes to the system could not be easily isolated. For instance, it was not uncommon that limitations of the component specification language required changing the back-end to add functionality needed for new kinds of components.

DiSTiL relies on IP for obtaining an abstract syntax tree representation of the specification, so it does not need a specialized parser. IP also provided a set of intentions implementing the C language on which DiSTiL was based. Additionally, the system relieved us from a series of low-level chores associated with managing source code. Another big gain was in the area of target code generation. The P2 component specification language (XP) had a generation scoping facility similar to that described in Section 2.2. XP, however, was severely limited in capabilities (being a token-based macro processor) and highly specific to the data structure domain (with built-in keywords like `container` and `cursor`). In contrast, generation scoping presents a domain-independent way to express generated code fragments and dependencies among them conveniently.

DiSTiL is an excellent demonstration of the benefits of using IP. DiSTiL's net development time was in the order of 9 man-months (excluding the development time for the generation scoping facility). The principal developer had not implemented a GenVoca generator before, so it is unlikely that this productivity was based on experience. A comparison with the P2 system may be useful, even though it is hard to compare directly the two generators (there are DiSTiL features that have no P2 counterpart and vice versa). We estimate that the P2 system required more than 3 man-years of work before it reached a level of functionality comparable to that of DiSTiL. The difference is reflected in the relative source sizes of the two generators. DiSTiL is about 10Kloc, with more than half of it (5.2Kloc) being components. P2, on the other hand, is not that much richer in components (17.1Kloc) but has an over 112Kloc infrastructure.

Domain-Specific Languages in Software Engineering. Domain-specific languages emphasize interface abstraction. This way, they can incorporate components that are uniformly treated and independently composable. Thus, the potential for software evolution becomes much greater. In DiSTiL this effect is obvious. Not only can the implementation of a primitive data structure change without affecting user code, but the specification of any user-defined data structure can also change without affecting the application. This is a characteristic of GenVoca generators. In the greater scheme of software engineering methods, GenVoca generators are compact representations of exponential-size libraries.

Programming in Domain-Specific Languages. In the evolution of languages, the development environments and supporting tools are at least as important as the merits of a language design. New languages require powerful compilers, good editing support, convenient debuggers, and useful libraries of components. This need is acknowledged in the design of IP. The system facilitates incremental changes both to the language and to the development environment. Thus, the cost of providing supporting tools for development in a new DSL is lowered.

Technical Advantages of Domain-Specific Languages. Domain-specific languages, as a means of software reuse, offer concrete technical advantages over standard static libraries (procedure libraries, object libraries, or binary component libraries). These mainly fall under the three categories of *simplicity*, *efficiency*, and *error detection*. A DSL can provide a uniform interface for different operations, thus simplifying its specification. Also, a DSL can often be more efficient than a static library: The flexibility of the DSL approach allows developers to experiment with different ways to express their requirements. Since a DSL implementation has access to high-level information, its optimizations are large scale (i.e., global program transformations) and often result in better code than that produced by a typical programmer. These benefits are well documented in the literature: The controlled-environment study of [Kie96] indicates productivity gain ratios of about 2.9 resulting from the use of generator technology. Our experience with P2 [Bat97b] showed performance gains of at least an order of magnitude in re-engineering a complex application.

Another advantage of the DSL approach is that it enables error checking that is more thorough and at a much higher level than that of general purpose languages. This enables error reporting that is both extensive and accurate [Bat97a], as in the case of DiSTiL. In contrast, traditional static libraries rely on the type checking facilities of their host language. Such mechanisms may be limited and, sometimes, almost non-existent: A well known disadvantage of STL is that C++ offers no mechanism to constrain a parameterization. As a result, invalid compositions are only detected well after composition time, yielding inaccurate and, depending on the implementation, possibly misplaced error messages.

Disadvantages of Domain-Specific Languages. The DSL approach has disadvantages. First of all, there is the non-trivial cost of designing and implementing a DSL. Assuming technical problems can be overcome and a DSL has been specified and implemented, a more serious obstacle emerges. The education cost associated with having developers use a new tool can be significant. This includes the reluctance of developers to abandon their favorite environments and learn to use new mechanisms (language, interpreter/compiler, debugger, etc.). If the DSL is simple and well supported, the problem is alleviated. The decision of adopting a DSL is based on the trade-off between the education cost and the expected benefits.

In the case of DiSTiL, a new element comes into play. The IP system helps reduce the education cost by letting the entire development environment adapt to a DSL. Making small, incremental changes to the programming environment (compiler, debugger) results in lower education costs than implementing a whole new system. There is a different price, however, that the user will have to pay. This is the one-time cost associated with learning to use the IP environment in the first place. The process involves a departure from text-based programming and the use of a structure editor, so the cost is not negligible. It will be interesting to see if the benefits of domain-specific extensions can offset this cost, making IP successful.

5 Conclusions and Further Research

DiSTiL is a software generator for the domain of data structures. It implements a powerful domain-specific language in a novel way: As an extension to the IP transformation system and using generation scoping — a general purpose meta-programming tool. We are interested in further exploring ways to facilitate implementing domain-specific languages. We assert that the existence of an extensible system in which new transformations can easily be specified is a very promising approach to designing domain-specific software. We believe that writing software generators as language extensions is a significant advancement that enables generator programmers to concentrate on the complexities of their task and not tedious infrastructure development. Hopefully, the future will bring more and more generators implemented as simple extensions and leveraging off common infrastructure.

Directions for future research include further development of a common ground for generator implementation — that is, a set of mechanisms widely applicable in generator writing. Such an effort will require careful modeling of generator activities and, particularly in the case of GenVoca, the interconnections between generator components. We expect this work to culminate to a collection of language primitives ideal for writing generators, in other words, a domain-specific language for implementing domain-specific languages.

6 References

[Bat88a] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C.Twichell, T.E. Wise, "GENESIS: An Extensible Database Management System", *IEEE Transactions on Software Engineering*, Vol. 14 #11 (November 1988), 1711-1730.

[Bat88b] D.S. Batory, "Concepts for a Database System Synthesizer", *ACM Principles Of Database Systems Conference* 1988, 184-192.

[Bat92] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components", *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat95a] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example From Avionics", *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.

[Bat97a] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997.

[Bat97b] D. Batory and J. Thomas, "P2: A Lightweight DBMS Generator", Accepted for publication in the *Journal of Intelligent Information Systems*, 1997.

[Boo87] G. Booch, *Software Components with Ada*, Benjamin/Cummings, 1987.

[Cli91a] W. Clinger and J. Rees. "Macros that Work". *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991, 155-162.

[Cli91b] W. Clinger and J. Rees (editors), "The Revised^4 Report on the Algorithmic Language Scheme". *Lisp Pointers IV(3)*, July-September 1991, 1-55.

[Coh89] D. Cohen, *AP5 Training Manual*. USC Information Sciences Institute 1989.

[Coh93] D.Cohen and N.Campbell, "Automating Relational Operations on Data Structures". *IEEE Software*, 10(3):53-60, May 1993.

[Cor91] J. Cordy, C. Halpern-Hamu and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", *Computer Languages* 16,1(Jan. 1991):97-107, also in *Proc. IEEE 1988 Intl. Conf. on Computer Languages*, 280-285.

[Die97] P. Dietz, C. Jervis, M. Kogan, and T. Weigert, "Automated Generation of Marshalling Code from High-Level Specifications", RNSG Research, Motorola, Schaumburg, Illinois, 1997./

[Due97] A. Van Duersen and P. Klint, "Little Languages: Little Maintenance?", *Proc. First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris 1997.

[Haa90] L. Haas, et al., "Starburst Mid-Flight: As the Dust Clears", *IEEE Transactions on Knowledge and Data Engineering*, March 1990, 143-151.

[Kie96] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A.Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, "A Software Engineering Experiment in Software Component Generation", *Fifth International Conference on Software Engineering*, 1996.

[Koh86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic Macro Expansion". In *Proceedings of the SIGPLAN '86 ACM Conference on Lisp and Functional Programming*, 151-161.

[Nei80] J. Neighbors, "Software Construction Using Components", Ph.D. Thesis, ICS-TR-160, University of California at Irvine, 1980.

[Per97] G. Jimenez-Perez and D. Batory, "Memory Simulators and Software Generators", *1997 Symposium on Software Reuse*, 136-145.

[Rea86] Reasoning Systems Incorporated, "REFINE User's Guide", Palo Alto, 1986.

[Rea89] Reasoning Systems Incorporated, "Dialect User's Guide", Palo Alto, 1989.

[Sim95] C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", *NATO Science Committee Conference*, 1995.

[Sim97] C. Simonyi, personal communication.

[Sir93] M. Sirkin, D. Batory, and V. Singhal, "Software components in a data structure precompiler". In *Proceedings of the 15th International Conference on Software Engineering*, May 1993.

[Sma96] Y. Smaragdakis and D. Batory, "Scoping Constructs for Program Generators". Technical Report TR-96-37, Department of Computer Sciences, University of Texas at Austin, December 1996.

[Ste90] G. Steele Jr., *Common Lisp: The Language*. Digital Press, 1990.

[Ste95] A. Stepanov and M. Lee, "The Standard Template Library". Incorporated in ANSI/ ISO Standards Committee C++ Draft.

[Vil97] E.E. Villarreal and D. Batory, "Rosetta: A Generator of Data Language Compilers", *1997 Symposium on Software Reuse*, 146-156.

[Wei93] D. Weise and R. Crew, "Programmable Syntax Macros". In *Programming Language Design and Implementation*, 1993, 156-165.

Appendix A: DiSTiL Realms and Components

Realm	Component	Description
Data Structures	Array	Random access array. Multi-dimensional in its general form but automatically specialized to fit the current specification without imposing run-time overhead.
	Dlist	Simple doubly-linked list.
	Hash	Hash table — re-implementation of the corresponding P2 layer.
	Odlist	Ordered doubly-linked list.
	Tree	Red-black tree (self-balancing binary tree). Code written to closely match the HP STL [HPSTL] implementation, more efficient than the AVL trees of P2.
Storage	Malloc	Elements are allocated on demand.
	Bounded	A maximum number of elements is pre-allocated. For cases when the maximum number of allocated elements is bounded.
	Persistent	Elements are stored persistently (on disk).
	Transient	Elements are stored in main memory (data structures don't outlive the process that created them).
Architectural	Functional	Forces code to be generated in functions to avoid source code bloating due to inlining.
DS Supplements	Delflag	Implements element deletion by marking them as deleted instead of de-allocating them.
	Sizeof	Keeps track of the data structure size.
	Check	Adds run-time bound checks.
	Inbetween	Ensures that a cursor always points to a valid element after deletions.
Hidden	Order	Factors out common code from all ordered layers. Inverts operations when the retrieval order is "decreasing" instead of "increasing".
	Outofbounds	Factors out common code from ordered layers. If a retrieval finds an element outside the range of a predicate, the search is complete.
	Predicate	Factors out common predicate handling code.
Special Purpose	Lrutree	A layer developed for a special-purpose application (LRU memory policy simulation). Implements a queue with special characteristics (quick position computation, move at top of queue).

Programming Language Support for Digitized Images or, The Monsters in the Closet

Daniel E. Stevenson*

Department of Computer Science
University of Iowa
Iowa City, IA 52242, USA

Margaret M. Fleck†

Department of Computer Science
University of Iowa
Iowa City, IA 52242, USA

Abstract

Computer vision (image understanding) algorithms are difficult to write, debug, maintain, and share. This complicates collaboration, teaching, and replication of research results. This paper shows how user-level code can be simplified by providing better programming language constructs, particularly a new abstract data type called a "sheet." These primitives have been implemented as an extension to Scheme.

Implementation of sheet operations is made challenging by the fact that images are extremely large, e.g. sometimes over 5 megabytes each. Therefore, operations that loop through images must be compiled from (a specialized subset of) Scheme into C. This paper discusses how the need for extreme efficiency affects the design of the user-level language, the run-time support, and the compiler.

1 Introduction

Within the past few years, computer imaging equipment has become dramatically cheaper and more reliable. Six years ago, a color camera and framegrabber cost $15,000 and was too heavy to lift. It can now be replaced by a $400 hand-held digital camera. As a result, cameras are becoming widely available. Both programmers and researchers are starting to incorporate images into computer science applications remote from the traditional home of images, computer vision (image understanding). The rapid spread of images is particularly obvious on the World-Wide Web.

Although many users require only image processing and graphics packages (e.g. xv, Photoshop), an increasing range of applications require basic image understanding. For example, researchers in the sciences and medicine use images to measure physical properties (e.g. blood vessel width) and screen for interesting events (e.g. unusual cell shapes). Companies, governments, non-profit organizations (e.g. museums), and private citizens are converting photo collections to digitized form. They require effective ways to locate images with specific content in large databases.

Computer vision lies on the border between computer science and electrical engineering. Traditionally, it has been isolated from the rest of computer science. In particular, it has seen little benefit from recent advances in the design and implementation of programming languages. Computer vision algorithms are still written primarily in C, occasionally now in C++.

As a result, computer vision code tends to be complex and repetitive. It is difficult to write, debug, maintain, and share. Inability to replicate results reported in research papers, which is the norm rather than the exception, is a major barrier to progress in this subfield. Collaboration with researchers in other areas of computer science is almost impossible.

More significantly, many computer vision algorithms are not much easier to program in high-level languages (in practice, always Common Lisp). Off-the-shelf compilers and interpreters do not provide the required efficiency. And the high-level language code tends to resemble a word-for-word translation of the C code. Existing abstract data types and con-

*Now at the University of Wisconsin–Eau Claire, Eau Claire, WI 54702

†Now at Harvey Mudd College, 301 E. Twelfth St., Claremont, CA 91711

trol constructs do not match the repetitive structure in this code and, therefore, cannot be used to simplify it.

This paper will present new abstract data types and associated operations. These primitives, implemented as an extension to Scheme [3], encapsulate much of the repetitive work common in computer vision code and can be compiled into efficient C code. We will discuss key issues involved in implementing the required compiler and run-time support, complementing previous work on compiling Scheme into C. And we will suggest how some of the features required in computer vision might find wider application in programming language design.

2 Images are huge!

The most distinctive feature of digitized images is their size. Digital cameras sold for the home PC market deliver 24-bit color images at sizes ranging from 320 by 240 up to 1600 by 1200. When the data is stored in packed binary arrays, this translates into between 0.23 and 5.76 **megabytes** per image. Images obtained from scanners or certain specialized cameras are even larger, as are video image sequences and 3D images from medical scanners (e.g. MRI). Although they can be compressed when stored in files, images must remain uncompressed during analysis.

Said another way, images are about 4-6 orders of magnitude larger than most objects traditionally found in a high-level language, and a single image may be comparable in size to the entire heap of a traditional Scheme session. Moreover, a typical vision function manipulates several such images (e.g. two inputs and one output). A typical interactive user doing experiments will use up all available memory in an attempt to manipulate (e.g. compare) as many images as possible simultaneously.

Because images are so large and many applications require fast response (e.g. tracking moving objects), computer vision programmers are obsessed with efficiency. Only the most stripped-down algorithms are fast when iterated over all one million locations in an image. Hand-coding critical functions in assembler has only recently become rare. This is a harsh environment in which to test programming language methods.

To avoid wasting scarce space, image values are typically stored as packed bytes, both in memory and in disk files, even though they are conceptually real numbers. Allocation and deallocation must be under user control, because images do not naturally become garbage (in the sense of becoming inaccessible to the user) quickly enough to prevent memory from filling up. They must be allocated outside the heap in a language such as Scheme, to prevent heap fragmentation and unnecessary copying of image data. And it must be possible for related images (e.g. an image and a subimage) to share storage.

Finally, algorithm designers must be extremely careful about how image data moves within the computer. Scanning through an image in the wrong order, on a machine with insufficient RAM, can cause dramatic swapping. Operations such as image rotation, which must access their input and output in different orders, sometimes require that very large images be divided into subblocks. Even if there is enough RAM, swapping could still occur inside internal memory caches.

When image processing or image-related graphics is done on an external board, transmitting image data between the board and main memory is often a significant fraction of total running time. Bus speed is one of the major obstacles in using PC-based cameras. On fine-grained parallel processors (e.g. the Connection Machine), transmission of the image from the camera to the processors may completely dominate running time.

3 Existing vision packages

A number of packages of standard image processing data structures and operations have been developed to aid research and teaching in computer vision. Some packages, such as the standard utilities xv and pbmplus, support only simple image processing and manipulation. Others, such as the HIPS Image Processing System [12], Khoros [10], LaboImage [6], and Matlab [19], cover a full range of low-level image processing utilities. Finally, some packages offer support for higher-level vision operations: the Image Understanding Environment (IUE) [11], KB-Vision [7], Vista [14], The Iowa Vision System [4], OBVIUS [5], and the Radius Common Development Environment [17]. Many large sites have at least one in-house package. And additional packages are used to support image understanding projects in scientific fields (e.g. astronomy, biology, medicine).

A typical computer vision package contains a library of C or C++ image operations and an interpreted front-end language. The front-end language is used to connect operations together, to communicate with the user, and to implement high-level ("smart") parts of algorithms such as automatic parameter setting and control of multi-stage or search algorithms.

3.1 Front-end language

Existing packages use a variety of front-end languages, whose main common feature is that they are always interpreted. Some use the Unix shell (HIPS) or interpreted C-like languages (Matlab). Others (Khoros, KBVision, IUE) use a graphical front-end language. Finally, one can use a high-level programming language, such as Common Lisp (Iowa Vision System, OBVIUS, Radius), Scheme, or ML. We strongly prefer to use a modern high-level language, because they are more powerful and simplify collaboration between computer vision and artificial intelligence.

The choice of front-end language is largely independent of which operations are included in the library. The Cantata dataflow interface has been used with at least three packages of operations (Khoros, KB-Vision, IUE). At least two recent languages, Tcl/Tk and the Elk [13] implementation of Scheme, were specifically designed to provide front-ends for a wide variety of applications.

3.2 Operations included in library

All reasonably-designed vision packages support a range of basic image manipulation operations such as display, cropping, rotation, altering greyscale or colormaps, and image statistics. Image processing packages (HIPS, Khoros, LaboImage, Matlab) also support linear filtering, some nonlinear filters, standard transforms (particularly the Fourier transform), and often simple edge detectors. However, these packages do not have good support for higher-level operations that manipulate edge-chains, features, and geometrical objects.

Packages designed for computer vision include data structures and pre-written code for high-level vision operations, either in the form of C++ classes and macros (IUE, KBVision, and Vista) or Common Lisp classes and methods (Iowa Vision System, OBVIUS, and Radius). However, the number of high-level operations in each package is quite limited. They typically include only one modern edge finder (typically Canny's), one camera calibration algorithm (invariably Tsai's), and a small selection of vision algorithms (edge segmentation and grouping, texture features, motion analysis, classification).

3.3 How have they fared?

Although these packages are all well-intentioned, and incorporate many interesting design ideas, there is no real prospect that any of them will become a standard tool used by most of the field. Computer vision algorithms are still the subject of active research and there are many recent, rival algorithms. However, each package includes only one or two algorithms for each task, often ones developed over ten years ago. For example, most packages do not include an edge finder more recent than Canny's.

Furthermore, different packages offer qualitatively different features. The best choice depends on how much money your site can spend, how much memory and disk space your machines have, what operating system you run, what applications you study (medical, satellite, etc.), what theoretical approach you favor, what other software (e.g. LISP) you have, and whether you prefer a graphic or a textual front-end.

In any attempt to address these problems, the most comprehensive packages (e.g. HIPS, IUE, KBVision) have grown so large that they are difficult to maintain and document. For example, the IUE contains over 575 classes, has over 800 pages of documentation, and consumes 500M of disk space [1]. A single package satisfying everyone's needs would be impossibly large.

4 A better approach

The difficulties in designing packages stem from the fact that designers are attempting to standardize at an inappropriate level of abstraction. Moreover, standardization at the correct level requires the full power of a compiler. Because there has been little cross-fertilization between computer vision and programming language design, package implementers have used only insufficiently powerful tools: library functions, classes, and macros.

4.1 The right level for standardization

Consider the case of the operators that generate texture features from a gray-scale image. Previous packages have attempted to provide a standard set of texture operators. However, the literature contains a wide range of texture operators, none of them have entirely satisfactory performance, respected researchers cannot agree about which ones perform best, and new operators are constantly being proposed. Since no consensus exists, standardization at this level is premature.

The correct place to standardize is at a level where there is a scientific consensus. This allows the programmer to have as much support as possible while making it easy to add new functionality and experiment with new variations in algorithms. Standardization at too low a level (C arrays) makes the programmer do most of the work by hand, while standardization at too high a level (image data structures, image processing functions) limits users to currently available techniques and discourages them from expanding the frontiers of science.

Therefore, we must provide standardization and support at the level of the basic data structures and control constructs used to write the library functions in computer vision packages. This would make the library functions simpler and easier to understand. Then, each programmer could create a package customized to their needs, by merging and modifying code from standard libraries.

4.2 Many drops of water make a river

Many previous researchers have approached this as a software engineering problem. Since each piece of repetitive work is conceptually simple, it should be easy for mature programmers to do it. So, it should be sufficient to standardize programming practice, so as to make everyone's code compatible. And, therefore, it should be sufficient to define a suitable set of macros, classes, and accessor functions (e.g. functions to extract value from different types of images).

This approach fails due to the sheer volume of pedestrian work required to properly write a low-level computer vision algorithm. Creative scientists, even junior ones and those doing applied industrial work, quickly get bored with repetitive coding. They will not take the time to make code sufficiently general or portable. And it is inappropriate to make them do so: repetitive work should be done by a computer.

4.3 The value of compilation

Our extension to Scheme, called Envision, uses a compiler to transform user-level Scheme code into efficient C code. Considerable research has been done on compilation of high-level languages and the newest Scheme-to-C compilers [9, 15, 16] perform quite well. However, these techniques have never been applied to generating computer vision code, partly due to lack of communication between programming language research and computer vision and partly due to the significant initial investment of time required to write or adapt a compiler.

Compilation offers several advantages in this domain. It allows library operations to be written in the same language used in the package front-end. Type inference can expand a small number of user-level type declarations into type assignments for all variables. We can efficiently implement a new control form (scan) which eliminates much of the work of looping through all locations in a 2D image, without requiring function calls inside these loops.

Finally, our compiler can automatically perform a variety of optimizations (section 9). Many of these optimizations are common in hand-written vision code. However, human programmers do not apply them consistently, they apply them in unsafe ways, they use approximations with poor numerical behavior, and they do not adapt quickly to changes in the hardware, operating system, or C compiler. It is safer and more efficient to centralize such knowledge in the hands of the compiler writer.

4.4 The necessity of compilation

It is tempting to think that the same effect could be achieved without writing a new compiler, by using facilities such as classes, macros, and type definitions (e.g. in ML). Unfortunately, current languages and compilers seem to lack the power required to define and optimize our new data structures and operations.

First, translating our high-level code into a standard language requires rewrite rules which operate at compile-time (so the output code is efficient) but which are type-dependent. At compile-time, Lisp and Scheme support only type-independent rewriting (macros). The type system in ML[20] seems to lack a mechanism for parameterizing a type by one or more numbers. This gives us no clean way to write a rule which manipulates objects of varying geometrical dimension but which requires access to their dimensions.

Second, a central feature of Envision is that missing (unavailable) values are first-class objects. For example, a variable which normally contains a real value may occasionally be assigned a missing value. Handling missing values requires modifications to type inference rules, modifications to standard arithmetic operations, code analysis to determine which expressions can never return a missing values, and restructuring expressions to minimize the number of tests for whether a variable value is missing. Standard compilers do not contain such support.

5 Sheets

The repetitive parts of low-level vision code can be isolated and removed from user-level code using a new data structure called a "sheet." Sheets represent the large areas of packed storage found inside image representations. They provide substantial capabilities beyond those of arrays, but only capabilities on which there is considerable consensus in computer vision. Using sheets, it is easy to construct any of the wide variety of image representations currently in use.

A sheet represents a function between two spaces: a set of locations (the domain) and a set of values (the codomain). Each space is locally Euclidean: every small neighborhood looks like a piece of $\mathbb{R}^n$ or $\mathbb{Z}^n$. The function is represented to finite precision: values are only available at a finitely dense set of locations and are only known with limited precision. Each sheet contains homogeneous data: all values have the same type and precision. This allows packed storage and optimization of compiled code.

Sheets provide a layer of abstraction which insulates the programmer from the details of how image data is stored in arrays. The sheet appears to contain data of the type described in mathematical specifications of the algorithm. For example, a log-polar stereo map might appear to be tubular, to contain signed floating point values given to the nearest 20^{th} of a unit, and to have no values for certain locations (e.g. where a surface was occluded in one image). The user need not know the details of how this is implemented using a standard array of unsigned 16-bit integers.

5.1 Features provided by sheets

Specifically, the sheets provide support for multi-dimensional values, arbitrary ranges of locations and values, continuity, user-specified precision, circularity, partiality, shared storage, and restrictions.

Multi-dimensional values: The domain of a sheet may be of any dimension. This capability is already provided by arrays. However, in addition, the values stored in a sheet may be points of any dimension. The current implementation supports 1D and 2D domains as well as 1D, 2D, and 3D values. There are several ways to simulate a multi-dimensional codomain using arrays: the programmer need not remember which method is being used.

The use of multi-dimensional values allows the user to represent a wide variety of image data structure with sheets. For example, a motion vector field can be represented using a 2D sheet with 2D codomain. The outline of a 2D image region can be represented using a list which contains, among other things, a continuous 1D sheet with 2D values (the x and y coordinates of the curve).

Range: The locations in a sheet may be any rectangular subsection of 1D or 2D space. For many applications, the origin of the coordinate system should be placed at the projection center of the image. By contrast, arrays force the origin to lie in one corner of the image, requiring geometrical algorithms to constantly add and subtract offsets.

Similarly, the user can freely specify what range of values will be stored in the sheet. The user is not confined to the limited selection of ranges provided by arrays (e.g. unsigned integers, signed longs, floats) nor does the range have to start at zero.

Continuity: The domain and codomain of a sheet may be either continuous (locally like $\mathbb{R}^n$) or discrete (locally like $\mathbb{Z}^n$). Images have a continuous domain and codomain. Edge maps have a discrete domain and codomain. Color maps have a discrete (1D) domain, but a continuous (3D) codomain.

Sheets with discrete domain provide values only at grid locations, whereas sheets with continuous domain provide values at any location within the bounds of the sheet (by interpolation). Sheets with continuous codomain provide values as real numbers, whereas sheets with discrete codomain provide values as integers.

Arrays, by contrast, always have a discrete domain. For the codomain, computer vision programmers are forced to choose between two bad options: discrete integers with an appropriate precision or continuous reals with an inappropriately high precision (thus wasting memory).

Precision: Numbers used in computer vision have a known, finite precision, due to a combination of quantization and contamination with random noise. When a sheet is created, the user specifies the precision of the values to be stored in it. This, together with the user's range specification, automatically determines the number of bytes used to store the value at each pixel. Similarly, the user specifies the spacing between pixel locations.

When using arrays, the spacing between pixel locations is always one unit. For the integer arrays typically used in computer vision, this is also true of the output values. This forces pyramid-based algorithms, for example, to explicitly rescale values.

Circularity: A sheet's domain and/or codomain may be circular. The current implementation supports the following *topological types*: linear (flat) domain and/or codomain, circular domain (e.g. a closed 2D region boundary), tubular 2D domain (e.g. a log-polar image), toroidal 2D domain (both dimensions are circular), and circular codomain (e.g. an image whose values are angles).

The topological type determines what happens if a program attempts to access locations outside the domain range, or store values outside the codomain range. For example, circular codomain values are reduced to the right range using modular arithmetic, whereas linear codomain values are approximated with the closest value in the range. Interpolation of circular values also uses modified algorithms.

Partiality: Values may be unavailable for some locations in a sheet. This may happen in the middle of the sheet (e.g. occluded regions in stereo disparity maps) or adjacent to its edges (e.g. an image which has been rotated or corrected for lens distortion). References to such a location, or to a location outside the sheet's storage range, return a special "missing" value.

In array-based code, missing values can be indicated by storing a special reserved value in the array. Unfortunately, programs don't all use the same reserved value. Different array types (e.g. unsigned short, signed long, float) require different reserved values. Most programs (notably edge finders) do not handle missing values at all.

Shared storage: The packed storage area of a sheet is separated from header information, such as scaling. Two sheets with different headers can share the same packed storage. As a result, rescaling or shifting a sheet does not require extensive memory allocation or copying, just creation of a new modified header.

Restrictions: The header information for each sheet includes a focus area, used to limit display and scanning operations (section 6.4). Thus, a subsection can be extracted from a sheet by combining a new header with the same packed storage.

5.2 How sheets improve programming

Certain packages provide support for some of these features, but this support is partial and erratic. As a result, most programmers build their own versions of features by hand. These implementations are special-purpose and incompatible with one another. They often use substandard methods, such as bilinear interpolation.

Standardized support allows simple and portable user-level code. It ensures that good methods are used for standard operations such as scaling and interpolation, that these operations are fully debugged, and that they are implemented using a standard portable language (e.g. ANSI C). It helps users write algorithms which handle the full range of images.

Figures 1 and 2 show C and Envision code for a typical operation. The Envision code is shorter than the C code, despite its longer (Lisp-style) function names. The C code is restricted to 8-bit unsigned values, whereas the Envision code handles sheets with any range of values. The Envision code uses second-order, rather than bilinear, interpolation and leaves smaller holes around missing values. It requires fewer input arguments. And, it rotates about the image center, a meaningful user-level location, rather than about the upper-left corner of the storage array.

```
void rewindow
  (double startx, double starty, double angle,
   *char array1, *char array2, int xsize1,
   int ysize1, int badval1, int xsize2, int ysize2,
   int badval2, int minval2, int maxval2)
{
   int newx, newy, lowx, highx, lowy, highy, intout;
   double realx, realy, errorx, errory, v1, v2, v3,
       v4, sinangle, cosangle, interpolated_value;
   sinangle = sin(angle); cosangle = cos(angle);
   for (newx = 0; newx < xsize2; newx++) {
    for (newy = 0; newy < ysize2; newy++) {
      realx = startx + newx*cosangle + newy*sinangle;
      lowx = floor(realx); highx = ceil(realx);
      errorx = (highx - realx);
      realy = starty + newy*cosangle - newx*sinangle;
      lowy = floor(realy); highy = ceil(realy);
      errory = (highy - realy);
      if (lowx < 0 || lowy < 0
            || highx >= xsize1 || highy >= ysize1)
          array2[(newx * ysize2) + newy] = badval2;
      else {
         v1 = array1[(lowx * ysize1) + lowy];
         v2 = array1[(lowx * ysize1) + highy];
         v3 = array1[(highx * ysize1) + lowy];
         v4 = array1[(highx * ysize1) + highy];
         if (v1 == badval1 || v2 == badval1
             || v3 == badval1 || v4 == badval1)
            array2[(newx * ysize2) + newy]
               = badval2;
         else {
            interpolated_value =
                 errorx * errory * v1 +
                 errorx * (1.0 - errory) * v2 +
                 (1.0 - errorx) * errory * v3 +
                 (1.0 - errorx) * (1.0 - errory) * v4
                 + 0.5;
            intout = interpolated_value;
            if (intout < minval2) intout = minval2;
            else if (intout > maxval2)
               intout = maxval2;
            array2[(newx * ysize2) + newy]
               = intout;}}}}}
```

Figure 1: C code to rotate and shift an image array.

6 Other new language features

To make full use of sheets, Envision provides a range of other new language features.

6.1 Points

A new data type, the "point," is introduced to represent locations in the domain of a sheet and values stored in its codomain. Following the standard conventions of pure mathematics, a 1D point is simply a number. Higher dimensional points, such as 2D and 3D points, are implemented as structures. Basic arithmetic operations are extended to work transparently on higher dimensional points.

```
(bulk-define
   rewindow ; name
  ((manifold 2 1) ; input types
       (manifold 2 1) real real real)
   unspecified ; output types
 (lambda (insheet outsheet xoffset yoffset angle)
    (let ((sinangle (sin angle))
          (cosangle (cos angle)))
      (scan (location outsheet)
         (let*
    ((point (sample->point location))
            (xcoord (point-coordinate point 0))
            (ycoord (point-coordinate point 1)))
         (sample-set! location
            (sheet-ref insheet
               (+ xoffset
                  (* xcoord cosangle)
                  (* ycoord sinangle))
               (- (+ yoffset (* ycoord cosangle))
                  (* xcoord sinangle)))))))))
```

Figure 2: Envision code for the same operation.

6.2 Missing values

Missing values (unspecified, bottom, ...) are widely used in programming language design. As far as we know, however, Envision is the first language in which they are first-class objects. That is, they can be assigned to variables, stored in data structures including sheets, and so forth. Basic arithmetic operations have been extended to handle the possibility that some of their inputs may be missing (typically returning a missing value) and to return missing values in other appropriate cases (e.g. division by zero).

We believe that first-class missing values are an extremely useful feature, which could be used elsewhere in high-level languages. For example, they could serve as the values of symbols which have not yet been bound, or as the value of assoc for items not in the list.

Missing values also represent a different philosophy for error handling. Standard languages trigger an error by default, and optionally let the user install error handlers. By default Envision triggers error breaks much more seldom. The user must force additional error breaks by explicitly testing whether some condition is satisfied (e.g. some value is nonmissing). This "mellow" convention is essential for image processing, in which the failed computation is typically only one of a million similar computations, most of which probably succeeded.

6.3 Samples

Any location in a continuous sheet can be accessed, but only certain specific locations, namely those on the storage grid, can be set. Envision includes direct pointers to grid locations, called "samples." Samples allow the programmer to bypass scaling and interpolation of locations in a sheet's domain, necessary for optimizing certain low-level vision algorithms.

To do this safely, programmers have no direct access to the raw array coordinates stored inside each sample. Rather, high-level operations allow them to find the sample nearest a floating-point location, retrieve the samples at the two extreme corners of a sheet's focus area, find the scaled coordinates of a sample, move by a specified displacement on the sample grid, and so forth. Because of the restricted direct access, Envision samples are similar to Java's "safe pointers."

6.4 Scanners

Applying a low-level vision operation to a sheet typically requires enumerating the samples in its focus area. In traditional computer vision programming, the user must extract the array bounds and compose a double loop. In Envision, most enumeration is done with the high-level primitive SCAN. Explicit loops are reserved for algorithms with unusual structure.

Scan has the following syntactic form, in which the test and the scanner are optional:

```
(scan (variable sample-or-sheet
       test scanner)
 expr1 expr2 ....)
```

Scan enumerates the samples in the focus area, binding the variable to each one in turn and evaluating the expressions inside the form. The scan starts at the input sample, or at the first location in the input sheet. When a sample passes the test or the end of the sheet is reached, it halts, returning the current sample and a boolean indicating whether it ran out of samples. The returned sample allows the scan to be restarted from where it left off.

The scanner input determines which samples are enumerated (e.g. every sample? every other sample?) and in which order. Envision includes a selection of standard scanners, including a default one used if the scanner input is omitted. Programmers can easily add new ones.

Scan differs from the standard Scheme map operator in two ways. It enumerates locations (samples), not values. This is essential in image processing, where an output value typically depends not only on the corresponding input value but also on values near it. Second, scan gives the programmer flexible control of the enumeration order. Such control is more important in computer vision than in traditional Scheme applications, because a 2D image can be ordered in more useful ways than a 1D list can.

6.5 Geometrical objects

Graphical display is largely handled by a single primitive DRAW. Input to DRAW includes a geometrical object, a window, a location in the window, and a list of options (e.g. color, filled, size). The type of the geometrical object determines what sort of figure will be drawn.

Geometrical objects include 2D sheets (mapped onto the window), 1D sheets (drawn as curves), points, line segments, polygons, ellipses, and text. The parameters in line segments, polygons, and ellipses may be points or 1D sheets. In the latter case, the object represents a set of objects if the sheets are discrete. If they are continuous, the object represents a swept strip or volume. These geometrical objects can also be used in implementing high-level vision algorithms.

6.6 File storage

Scheme includes only ASCII file I/O, unsuitable for storage of objects as large as sheets. Envision provides a second type of structured file I/O, in which packed sheet data is written in binary and other objects are written in a tagged ASCII representation. Operations are also provided to read bytes and sequences of bytes, for reading other image file formats (e.g. PPM, JPEG) and communicating with devices (e.g. cameras).

6.7 Storage areas, open files, windows

As described above (section 2), sheets cannot be garbage collected automatically. To help the user manage sheet space, Envision maintains a list of *storage groups*, i.e. sets of sheets sharing a common packed storage area. For each storage area, it can provide one representative sheet, which the user can examine, e.g. when deciding whether to deallocate it.

Since we are forced to provide storage-management tools for one badly-behaved type of data, there is no conceptual problem extending such tools to other places where they would be useful. The ease with which users can lose pointers to open files and windows is a long-standing problem in high-level language design. Envision allows the user to list such open connections.

7 Our implementation

We have implemented Envision as an extension to the Scheme48 implementation of Scheme [9]. The overall structure is determined by three main ideas: separation of sheet data from the high-level front-end, variable compilation, and separation of sheet type handling into run-time and compile-time components.

7.1 Two-processor architecture

We have implemented Envision using two processes. The front-end is Scheme, augmented with a variety of functions implementing Envision's user front-end and its compiler. Sheet data, window graphics, and binary file I/O, however, are handled by a separate C program. This "coprocessor" is connected to Scheme via a Unix socket.

Crucially, packed data from sheet storage areas is never passed down the socket. To the user, sheet data appears to reside in Scheme. In fact, the packed storage areas for sheets reside in the coprocessor. Scheme has only the header data for each sheet, plus a unique identifier that allows it to identify the storage area when communicating with the coprocessor. Therefore, the socket connection does not have to be fast. In our experience, the only interfaces fast enough for image transmission use shared memory and these tend to be fragile.

This architecture offers three advantages. First, since our primary field is not programming languages and we wanted to produce a usable prototype quickly, we simplified our work by using an existing Scheme implementation and not modifying its internals. Second, individual users create new coprocessor binaries whenever they link in C code generated for them by the compiler. It is convenient that they need only rebuild the coprocessor binary, because the Scheme binary is 30 times larger and more complicated to rebuild. Finally, we wanted to test how well algorithms could be divided into sheet-processing and high-level parts. A clean algorithmic separation would simplify taking advantage of specialized image processing boards or a second processor, without placing undue strain on the bus or network connecting these to the main processor.

7.2 Variable compilation

A typical computer vision algorithm contains both functions that manipulate the values within sheets and high-level functions that operate on more traditional data structures. Because they manipulate objects of grossly different size, these two types of functions require very different types of compilation. (It would be premature to decide whether this is a binary distinction or two samples from a continuous variation.)

High-level scheme functions are compiled into Scheme48's byte code by Scheme48's compiler. They can use all the features of Scheme and Envision. However, Scheme functions which use sophisticated features cannot be compiled into code efficient enough to scan operations across large sheets, on current hardware. [1]

Functions which scan across sheets must be compiled for maximum efficiency. The types of all variables must be determined at compile time. The user must declare the types of input and output values, because it is frequently impossible to infer whether numbers and sheets are real/continuous or integer/discrete. An error is triggered if the compiler cannot determine the type of any internal variable.

Furthermore, such functions cannot allocate or dellocate non-stack space, nor can they call graphical display functions. Variables are restricted to points, booleans, sheets, and samples. Only points can be

[1]They would run faster if compiled into C, e.g. using Bigloo [15, 16], but not fast enough.

missing. Only basic control constructs are allowed. These restrictions were inspired, in part, by those of Prescheme [9].

Previous optimizing Lisp and Scheme compilers, such as Bigloo or Franz Common Lisp, regard the input code as fixed and take as their goal optimizing it as much as possible. The harsh environment of image processing forces us to take a different attitude for sheet-handling code: the code must run with sufficient speed and therefore the user must be helped to write such code, by a combination of language restrictions and compiler errors. Computer vision researchers find it frustrating to guess what modifications to their code will convince a general-purpose compiler to make it run fast enough.

Envision's compiler produces two types of output for sheet-handling functions. For debugging code on small images, they can be compiled into code that runs on the coprocessor's stack machine (section 8.2). For final use, they are compiled into C code, which can be linked directly into the coprocessor.

7.3 Sheet typing

Type features for sheets (and, by extension, samples) are divided into two groups. The dimensionality of the domain and codomain, and whether each is continuous or discrete, have profound effects on algorithm design. Only the most trivial functions (e.g. copy) are polymorphic across these distinctions. Furthermore, they drastically affect C code generation, e.g. whether C variables are declared as floats or ints, whether 1D or 2D code is substituted when expanding a SCAN form, and whether the inputs to addition are numbers or vectors. Therefore, these type distinctions are resolved at compile-time.

By contrast, the design of most computer vision algorithms does not depend on the other type features: range, precision, circularity, or whether missing values might be present. Previous systems typically forced users to make some of these distinctions at compile-time, resulting in annoyingly non-general code (e.g. edge finders that would run only on signed 8-bit images). The increase in generality obtained by resolving these distinctions at run-time is worth the small penalty in running time.

8 The Envision coprocessor

The coprocessor provides four capabilities: allocation and dellocation of packed storage for sheets, the run-time component of sheet support, a stack machine that can run user-defined code, and a graphics interface with a built-in event loop.

8.1 Implementation of sheets

Sheets are implemented as arrays of bytes. Depending on the range and precision requested by the user, between one and three bytes are allocated per pixel. Missing values are marked by storing a special reserved value into the array. Sheets with 2D domain are implemented using a 1D array of pointers to the first element in each row, which results in faster access than standard address arithmetic. Sheets with 2D or 3D codomain are implemented using two or three arrays.

With each sheet, the coprocessor stores three accessor functions. These accessors are given raw array coordinates: scaling is handled by Scheme and the compiler. Two accessors retrieve and set the value at a particular integer array location. The third accessor, present only for sheets with continous domain, retrieves an interpolated value for a location specified by floating point coordinates. All three accessors handle circularity, missing values, and range restrictions. However, they return raw integer values, to which the compiler must apply the appropriate scaling and offset.

Interpolation is implemented using a nine-point second-order polynomial interpolate. When all 9 values are available, this is similar to the six-point interpolate described in [2]. However, our interpolation algorithm handles any pattern of missing values, handling simple cases quickly and decaying gracefully to a bilinear, linear, or nearest-neighbor interpolate as required. This capability is required for correct, fully general implementation of operations such as image rotation. However, it would be very difficult for most users to implement on their own and no previous vision package provides it.

8.2 Running functions

From the Scheme interpreter, the user can call user-defined functions installed in the coprocessor. These include fully compiled functions linked directly into the coprocessor and also functions compiled, for debugging, into instructions for the coprocessor's stack interpreter. The interpreter includes simple assembler instructions, a function calling mechanism, special handlers for basic sheet operations, and a wide range of mathematical functions directly available in the standard C library (e.g. trignometric functions).

When some inputs to a function are sheets, the sheet header information is passed from Scheme to the coprocessor. Sheet headers are large compared to the other types allowed (216 bytes), and it is very inefficient to copy them around C's stack or the coprocessor's stack. Therefore, sheet headers passed from Scheme are stored in a special array and referred to by number. The contents of this array are static during the function call, because user-defined coprocessor functions cannot create, delete, or modify sheets.

8.3 Graphics support

Finally, the coprocessor manages the user's interaction with the window system. It includes operations for creating and destoying windows, and primitives for displaying graphical objects on them. Events such as window resizing, motion, and exposure are handled automatically. Command-type mouse events (moving and clicking inside windows) will eventually be handled by an X-based user-interface program. The current implementation uses the X window system but the user-level language is generic and should be compatible with many window systems.

9 The compiler

The Envision compiler transforms the user-level code into an intermediate language, with operations and control structure similar to C, but with Lisp-like syntax. This transformation is carried out by Scheme-to-Scheme transformation rules inspired by (but rather different from) those in [8]. From the intermediate language, it is easy to generate both C code and code for the coprocessor's stack machine.

With the exception of certain straightforward basic components, our compiler handles problems largely disjoint from those treated by previous compilers (e.g. [9, 15, 16]). On the one hand, we excluded control constructs which threatened to create difficulties and which did not seem useful in writing sheet-handling functions (which tend to have a restricted structure). On the other hand, the Envision compiler spends considerable effort optimizing the handling of higher-dimensional objects and missing values.

9.1 Type inference

Envision's type inference engine allows types to be parameterized by dimension. Points have a single dimension parameter (1D, 2D, or 3D). Sheets and samples have two dimensions: one for the domain and one for the codomain. This allows type inference rules to enforce appropriate dimension relations among the inputs to a function, and between the inputs and outputs. It also gives compiler functions access to the dimensions as numbers. For example, the function which extracts the kth coordinate of a vector must test whether k is in the correct range.

Type inference handles missing points by assigning them a type with a special (wildcard) value in place of the dimension. These types are fused with normal types in the obvious way, e.g. when a missing and a non-missing value are generated by the two branches of an if statement.

9.2 Scanner substitution

To avoid function calls inside loops at run-time, the enumeration code from the specified scanner is substituted into each scan form at compile time. This substitution process must examine whether the sheet input is a sheet or sample, and whether it is 1D or 2D. For example, the default scanner FORWARD contains two versions of the looping code, one for 1D sheets and one for 2D sheets. While not conceptually complex, type-dependent substitution cannot be accomplished with standard Scheme macros and must therefore be built into the compiler.

9.3 Real numbers

It is incredibly inefficient to rebuild real-number handling, including utilities such as trigonometric functions and random-number generation, on top of integer arithmetic. C provides well-optimized (if not ideally general) real number handling, which we used.

9.4 Missing values

To a first approximation, a missing value might appear anywhere that a point is expected. A naive implementation might incorporate a test for missing values into the implementation of each sensitive operation, e.g. all the numerical functions. However, this results in massively more testing than is performed in hand-written code.

The compiler performs a "purity analysis" to determine which variables and expressions can never return a missing result. A value may never be missing because it came (directly or indirectly) from a constant input. It might be the result of a strict operator [2] applied to never-missing values. Certain operations for retrieving sheet parameters (e.g. the offset) cannot return missing values. These appear in the expansion of extremely common functions, notably those for setting and accessing values in sheets.

Finally, a value can never be missing if it is protected by an explicit test for whether it is missing. Some tests occur in the user-level code. Others are generated automatically when the compiler expands sensitive operations.

The compiler also recognizes connected groups of strict functions and collects all required tests for missing values at the start of the group. Prior to this step, the compiler restructures the code so that expressions do not contain any forms which might generate side-effects, so that a left-to-right order of evaluation is preserved when sub-expressions are extracted.

9.5 Vector operations

Expansion of vector operations ensures that temporary variables are allocated as needed, but not if the input is a variable or a constant. Moreover, if an input is explicitly headed by a 2D or 3D point constructor (or a sequence ending in such a form), the input is decomposed into its constituents at compile-time rather than at run-time. For example, a sequence of 3D vector + and * operations will become three 1D numerical expressions followed by only one point constructor.

This complex analysis of the input, and deconstruction of some inputs, is beyond the power of restricted macro facilities such as Scheme's define-syntax. Moreover, the proper expansion depends on the dimensionalities of the inputs and, thus, it must follow type inference.

[2] I.e. an operator which never returns a missing value when given non-missing inputs.

Expansion of vector operations must follow generation of tests for missing inputs. The vector expansions destroy groups of strict operations, because a single user-level operation may become a complex form which sets and uses temporary variables. (A particularly bad example is the cross-product operation.) Therefore, even if a general-purpose compiler supported missing values, it would still be difficult to define vector operations using user-level features such as macros or classes.

9.6 Sheet references

Operations which set or access sheet values must also be expanded with some care. To a first approximation, they simply add scaling and offset to the low-level access function. However, like vector operations, they must allocate temporary variables exactly when needed. Furthermore, if the domain and/or codomain is discrete, scaling must be done using exact integer operations which check that the divisions involved have no remainder. Therefore, certain parts of this expansion must be delayed until after type inference has been done.

9.7 Space allocation

Dynamic storage allocation is too slow to be allowed inside sheet-handling functions. All the non-stack storage required must be allocated before the function is called, and passed to it. Therefore, sheet-handling functions cannot use lists to return several objects and so the compiler must support multiple return values.

Points, including numbers and missing points, are represented by C structs. When such a value is returned by a form inside a SCAN loop, it is inefficient to create the struct anew in each iteration of the loop (even on C's stack). Instead, the compiler reserves stack space for all such internal point structs once, at the start of the sheet-handling function.

When, in addition, an internal point value is known never to be missing, the compiler can pre-set the missing? field of the point struct. Thus, inside a scan loop, the only fields of the struct which must be examined or set are the actual numerical values.

This optimization eliminates essentially all overhead on 1D arithmetic involving non-missing numbers.

Attempting to avoid overhead on simple arithmetic is common in compilers for high-level languages. However, it is difficult to do in the context of a form which is evaluated only once (or a few times) each time the function is called. Our optimization is easy and often applicable, because it exploits the fact that scan loops contain many strict arithmetic operations and forms inside scan loops are evaluated many times.

9.8 Linear operations

Hand-written C code often takes advantage of the fact that image processing operations often add or subtract values from different locations in the same image. We intend, eventually, to detect forms making multiple references to the same sheet and factor out the scaling and offset applied when extract values from the sheet.

10 Implementation results

We are in the late stages of debugging Envision and writing appropriate example code and new-user documentation. Draft documentation can be found on our web site [18] and we anticipate releasing the code in early fall. It requires only standard Unix components (ANSI C, sockets, and Xlib) and currently runs on three operating systems (Linux, HP-UX, and SGI Irix). Both Scheme48 and Envision are extremely small systems: compressed source for both will fit on two floppy diskettes.

The current system is fast enough for many types of image processing research. Rotation of a 538 by 364 image (by an arbitrary angle) takes 6 seconds on a 120MHz Pentium PC and under 3 seconds on an SGI Indigo 2. Functions run on the coprocessor's stack machine take about 7 times as long: still sufficiently fast for debugging using small examples. We believe that a combination of further optimizations and rapidly increasing processor speed will eventually make the output suitable for a wide variety of image processing applications. The most demanding real-time applications may, however, require a more sophisticated compiler which can take advantage of image processing boards.

11 Conclusions

This paper has shown how techniques of modern programming language design can be successfully applied to an unusual domain: computer vision. We have seen that that image handling requires interesting new programming language constructs, including new data types (sheets, samples), a new mapping-type operator (scan), and first-class support for missing values. We have also seen that, although the huge size of images demands extreme efficiency, this can be achieved by compiling a high-level language using current methods, appropriately adapted to the specific application.

Acknowledgments

This research was supported by NSF grants IRI-9501493, IRI-9420716, and IRI-9209728. We would also like to thank Richard Kelsey, Jonathan Rees, Manuel Serrano, and Val Tannen.

References

[1] Amerinex Applied Imaging (1996) "Image Understanding Environment Program: Overview," http:// www.aai.com/ AAI/ IUE/ IUE.html, 9 September 1996.

[2] Bracewell, Ronald N. (1995) *Two-Dimensional Imaging*, Prentice-Hall, Englewood Cliffs NJ.

[3] Clinger, William, Jonathan Rees, et al. (1991) "Revised^4 report on the algorithmic language scheme," *ACM Lisp Pointers* 4/3, 1–55.

[4] Fleck, Margaret and Daniel Stevenson, "The Iowa Vision System Project," http://www. cs.uiowa.edu/ ~mfleck/ vision-html/ vision-system.html, 13 September 1996.

[5] Heeger, David and Eero Simoncelli, "OBVIUS," http:// white.stanford.edu/ ~heeger/ obvius.html, 1 November 1996.

[6] Jacot-Descombes, Alain, Marianne Rupp, and Thierry Pun, "LaboImage: a portable window-based environment for research in image processing and analysis," *SPIE Proc. Vol 1659: Image Processing and Interchange: Implementation and Systems* (1992), pp. 331–340.

[7] "KBVision," Amerinex Applied Imaging Inc., Amherst, MA, http:// www.aai.com/ AAI/ KBV/ KBV.html, 1 November 1996.

[8] Kelsey, Richard and Paul Hudak (1989) "Realistic Compilation by Program Transformation," *Proc. 16th ACM Symp. on Principles of Programming Languages*, pp. 281–292.

[9] Kelsey, Richard and Jonathan Rees (1995) "A Tractable Scheme Implementation", *Lisp and Symbolic Computation 7(4).*

[10] "Khoros," Khoral Research Inc., Albuquerque, NM, http:// www.khoros.unm.edu/ khoros/, 1 November 1996.

[11] Kohl, Charles and Joe Mundy (1994) "The Development of the Image Understanding Environment," *CVPR 94*, pp. 443–447.

[12] Landy, Michael S., Yoav Cohen, and George Sperling, "HIPS: A Unix-Based Image Processing System," *CVGIP 25* (1984), pp. 331–347.

[13] Laumann, Oliver and Carsten Bormann (1994) "Elk: the Extension Language Kit," *USENIX Computing Systems* 7/4, pp. 419-449.

[14] Pope, Arthur R. and David G. Lowe, (1994) "Vista: A Software Environment for Computer Vision Research," *CVPR 94*, pp. 768–772.

[15] Serrano, M. and Weis, P. (1995) "Bigloo: a portable and optimizing compiler for strict functional languages," SAS 95, pp. 366–381.

[16] Serrano, M. (1994) *Vers une compilation portable et performante des langages fonctionnels*, Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France.

[17] SRI International, "RCDC Home Page," http:// www.ai.sri.com/ perception/ software/ rcde.html, 1 November 1996.

[18] Stevenson, Daniel and Margaret Fleck, "Envision: Scheme with Pictures," http://www. cs.uiowa.edu/ ~mfleck/ envision-docs/ envision.html, 7 June 1997.

[19] Thompson, Clay M. and Loren Shure (1993-95) "Image Processing Toolbox for use with MATLAB," MathWorks Inc., Natick, MA.

[20] Ullman, Jeffrey D. (1994) Elements of ML Programming, Prentice-Hall, Englewood Cliffs NJ.

Modeling Interactive 3D and Multimedia Animation with an Embedded Language

Conal Elliott
Microsoft Research

http://www.research.microsoft.com/~conal

Abstract

While interactive multimedia animation is a very compelling medium, few people are able to express themselves in it. There are too many low-level details that have to do not with the desired content—e.g., shapes, appearance and behavior—but rather how to get a computer to present the content. For instance, behaviors like motion and growth are generally gradual, continuous phenomena. Moreover, many such behaviors go on simultaneously. Computers, on the other hand, cannot directly accommodate either of these basic properties, because they do their work in discrete steps rather than continuously, and they only do one thing at a time. Graphics programmers have to spend much of their effort bridging the gap between what an animation is and how to present it on a computer.

We propose that this situation can be improved by a change of language, and present *Fran*, synthesized by complementing an existing declarative host language, Haskell, with an embedded domain-specific vocabulary for modeled animation. As demonstrated in a collection of examples, the resulting animation descriptions are not only relatively easy to write, but also highly composable.

1 Introduction

Any language makes some ideas easy to express and other ideas difficult. As we will argue in this paper, today's mainstream programming languages are ill-suited for expressing multimedia animation (3D, 2D and sound), both in their basic paradigm and their vocabulary. These languages support what we call "presentation-oriented" programming, in which the essential nature of an animation, i.e., *what an animation is*, becomes lost in details of *how to present it*. We consider the question of what kind of language is suitable for capturing just the essence of an animation, and present one such language, *Fran*, synthesized by complementing an existing declarative "host language", Haskell, with an embedded domain-specific vocabulary.

We propose an alternative to "presentation-oriented" programming, namely "modeling", in which a model of the animation is described, leaving presentation as a separate task, to be done automatically. This idea of modeling has been applied fruitfully in the area of non-animated 3D graphics as discussed below, and is now almost widely, though not universally, accepted. Our contribution is to extend this idea in a uniform style to encompass as well sound and 2D images, and across the time dimension, in order to model animations over a broad range of types. For brevity, this paper concentrates on 3D animation, but it is really the uniform integration of different types that gives rise to great expressive power. (See Elliott and Hudak [1997] for 2D examples.)

While imperative programming languages are suited to presentation-oriented programming, the modeling approach requires a different kind of language. Unfortunately, bringing a useful new language into being is quite a daunting task, requiring design of semantics and syntax, implementation of compilers and environment tools, and writing of educational material. However, as Peter Landin taught us thirty years ago, we can logically separate a language into (a) a domain-specific vocabulary and (b) a domain-independent way of composing more complex things from simpler ones. In other words, a language is a combination of a "host language" and a "domain-specific embedded language" (DSEL). By reusing the same host language for several different vocabularies, we can amortize the cost of its creation over more uses. In fact, unlike thirty years ago, we are now fortunate enough to have various candidate languages from which to choose. In this paper, we examine various features of a candidate host language to see which are helpful and which are not helpful for modeled animation. We find that Haskell is a fairly good fit, requiring only a few compromises.

The rest of this paper is organized as follows. Section 2 starts with a few examples of modeled animations. Section 3 introduces the notions of presentation and modeling for non-animated 3D graphics, and looks at some concrete benefits. Section 4 extends the idea and benefits of modeling to a variety of types besides 3D geometry, including sound and 2D images, and across the time dimension. Section 5 considers the pragmatics of creating a new domain specific language (DSL), and motivates the DSEL approach. Section 6 examines the usefulness of host language features in some detail. The remainder of the paper looks at related work and describes some directions for future work on modeled animation.

2 Examples

In this section we present a handful of modeled animations, in order to make later discussion more concrete.

2.1 Static models

To start, we import a simple 3D model of a sphere from "X file" format.

```
sphere :: GeometryB
sphere :: GeometryB
sphere = importX "sphere1.x"
```

The type `GeometryB` represents 3D geometry animations. (This "animation" happens to be a "static", i.e., not time-varying, one.) Similarly, we import a teapot model. However, the teapot is in an awkward orientation, so we adjust it after importing, rotating around the X axis by an angle of $-\pi/2$:

```
teapot :: GeometryB
teapot =
 rotate3 xVector3 (-pi/2) **%
 importX "tpot2.x"
```

Note that in Haskell, function application binds more tightly than all infix operators. Here are the types of the modeling vocabulary we used:

```
xVector3 :: Vector3B
rotate3  :: Vector3B -> RealB
            -> Transform3B
(**%)    :: Transform3B -> GeometryB
            -> GeometryB
```

The constant `xVector3` is the unit vector pointing in the positive X direction. `rotate3` takes an axis vector and a number and yields a 3D transform. The operator `**%` applies a 3D transform.

2.2 Spinning

Although types like `GeometryB` and `Vector3B` are potentially animated, the example so far uses static animations. Next we will color the teapot red and make it spin around the Y axis.

```
redSpinningPot =
  rotate3 yVector3 time **%
  withColorG red teapot
```

The new features are "time", the unit Y vector and application of a color to a geometric model:

```
time :: RealB
yVector3 :: Vector3B
withColorG :: ColorB -> GeometryG
              -> GeometryG
```

The use of `time` here deserves special attention. It is a primitive number-valued animation (hence the type `RealB`) representing the flow of time. Note that `time` is not a mutable real value, but a fixed animation. Animations are essentially functions of time, with `time` being the identity function, and operations like `rotate3`, `withColorG`, being `**%` are combinators that map functions of time to functions of time.

2.3 Generalizing

Next, generalize this simple spinning teapot, so that its color and rotation angle are parameters.

```
spinPot :: ColorB -> RealB -> GeometryB

spinPot potColor potAngle =
  rotate3 yVector3 potAngle **%
  withColorG potColor teapot
```

We will make use of the `potSpin` function in a series of three interactive 2D animations.

```
spin1, spin2 :: User -> ImageB
spin1 = withSpin potSpin1
spin2 = withSpin potSpin2
```

When an animation is interactive, its type is a function from the user supplying input. Hence the type above. Yet to be defined are `withSpin`, `potSpin1`, and `potSpin2`. First, we will give their types and an informal description of their purpose.

```
potSpin1, potSpin2 :: RealB -> User
                      -> GeometryB

withSpin :: (RealB -> User -> GeometryB)
         -> User -> ImageB
```

The two `potSpin` functions take as arguments an animated number, which will be related to the rotation angle passed to `spinPot`, and a user from which to get input. In the simplest case, just ignore the user, use red for the pot color, and pass on the angle argument unchanged:

```
potSpin1 angle u = spinPot red angle
```

The `withSpin` function takes one of these geometry producers and renders it together with some textual instructions.

```
withSpin f u =
  growHowTo u `over`
  renderGeometry (f (grow u) u)
                 defaultCamera
```

The function grow will be defined below. Its job is to turn user input into an animated angle, which gets passed to the geometry producer. The produced geometry is rendered with a default camera to produce a 2D animation, which is combined with the instruction text image. The function renderGeometry takes geometry and camera (animated as always), and yields a 2D animation:

```
renderGeometry :: GeometryB
               -> Transform3B -> ImageB
```

2.4 A more interesting pot spinner

Before looking into the definition of `grow`, we will see the second pot-spinning geometry producer, which adds a few new features:

- A light source is added and, visualized as a white sphere that orbits the spinning teapot. For convenience, the translation vector is specified in spherical coordinates.
- The teapot's color is animated, and specified in HSL coordinates.
- The "angle" argument generated by `grow` and passed by `withSpin` is integrated, and so is interpreted as the rate of change of the angle.

The definition:

```
potSpin2 potAngleSpeed u =
  spinPot potColor potAngle `unionG`
light
   where
    light = rotate3 yVector3 (pi/4)    **%
            translate3 (vector3Spherical
                           2 time 0) **%
            uscale3 0.1               **%
            withColorG white (
             sphere `unionG` pointLightG)
    potColor =
       colorHSL (sin time * 180) 0.5 0.5
    potAngle = integral potAngleSpeed u
```

Note the expression "`sin time * 180`" used in defining the teapot's color. The meaning of `sin` and "`*`" are not the usual ones, operating on numbers, but rather counterparts "lifted" to consume and produce number-valued animations (of type `RealB`). Even the numeric literal `180` is taken to mean an unchanging number-valued animation (having type `RealB`). Haskell's overloading ability, based on type classes is responsible for this great syntactic convenience. Several dozen functions have been lifted in this way, so that, for instance, `sin` and "`*`" not only have the usual types

```
sin :: Float -> Float
(*) :: Float -> Float -> Float
```

but also

```
sin :: RealB -> RealB
(*) :: RealB -> RealB -> RealB
```

2.5 Reactive growth

Now we turn to `grow`, which converts user input to a time-varying angle (of type `RealB`). It is defined as the integral of the value generated by `bSign`, defined below, which produces an animated number that has value zero when no mouse buttons are pressed, but switches to negative one or positive one while the user is holding down the left or right mouse button. The angle value produced by `grow` is thus growing while the right button is pressed, shrinking while the left is pressed, and constant when neither button is pressed.

```
grow :: User -> RealB

grow u = integral (bSign u) u
```

(The reason that even `integral` takes a user argument is that integration is done numerically, and must somehow know how hard to work on the approximation.)

The `bSign` function is itself defined in terms of a more general function `selectLeftRight`, which switches between three values, depending on the left and right button states.

```
bSign :: User -> RealB
bSign u = selectLeftRight 0 (-1) 1 u

selectLeftRight :: a -> a -> a -> User
                -> Behavior a

selectLeftRight none left right u =
  condB (leftButton u)
        (constantB left)
     (condB (rightButton u)
            (constantB right)
        (constantB none))
```

Some explanation: the use of a lower-case type name ("a") above means that `selectLeftRight` is polymorphic, applying to any type of argument. The function `condB` is a behavior-level conditional, taking an animated boolean and two animated values, and choosing between the two continuously. The Fran primmitive `constantB` turns a regular "static" value into a constant animated value (as required here by `condB`). The `leftButton` and `rightButton` functions tell whether the mouse buttons are pressed.

It is easy to define these two button state functions, in terms of a toggling function that takes an initial value and two events that tell when to switch to true and when to false.

```
leftButton, rightButton ::
  User -> BoolB
leftButton  u = toggle (lbp u) (lbr u)
rightButton u = toggle (rbp u) (rbr u)

toggle :: Event a -> Event b -> BoolB
toggle go stop =
  stepper False (  go   -=> True
               .|. stop -=> False)
```

The functions lbp, lbr, rbp, and rbr, yield left and right button press and release events.

```
lbp, rbp, lbr, rbr :: User -> Event ()
```

The `stepper` function takes an initial value *v* and an event *e*, and yields a piecewise-constant behavior that starts out as *v* and switches to the values associated with occurrences of *e*. In the definition of `toggle`, the event is constructed from the `go` and `stop` argument events, using the event handling operator "**-=>**" and the event merging operator "**.|.**". As a result, the constructed event occurs with value `True` whenever go occurs and with value False whenever `False` occurs. (Note: the event operators are described in Elliott and Hudak [1997], but their semantics have changed since that publication, and now consist of a *sequence* of occurrences, not just a single one. Also, the button press events and mouse motion behavior are functions of a `User` rather than a start time.)

2.6 Adding instructions

Finally, to produce instructions and user feedback, we define `growHowTo`, which produces a rendered string, colored yellow and moved down to be out of the way. The text gives instructions when neither button is pressed, says "left" while the left button is pressed, and "right" while the right button is pressed. Its definition involves 2D versions of vectors, transform formation and application, and coloring, plus the polymorphic function `selectLeftRight`, defined above.

```
growHowTo :: User -> ImageB

growHowTo u =
  moveXY 0 (-1) (
   withColor yellow (
    simpleTextImage messageB ))
 where
   messageB =
     selectLeftRight
       "Use mouse buttons to \
            \control pot's spin"
       "left" "right" u
```

Many more examples of functional animation may be found in Elliott and Hudak [1997], Elliott [1997], and Daniels [1997]. See also the user's manual (Peterson and Ling [1997]), which contains precise types and informal meanings of the embedded animation modeling vocabulary and still more examples.

With the given examples in mind, we step back from our chosen approach to expressing interactive animation, and consider the history, the benefits of "modeling", and of language embedding.

3 Presentation vs. modeling for 3D geometry

The practice of 3D graphics programming has made tremendous progress over the past three decades. Originally, if you wanted your program to display some graphics you had to work at the level of pixel generation. You had to master scan-line conversion of lines, polygons, and curved surfaces, hidden surface elimination, and lighting and shading models—rather complex tasks. A significant advancement was the distillation of this expertise into rendering libraries (and of course underlying hardware). With a rendering library, such as GL by Silicon Graphics, you could express yourself at the level of triangles and transformation matrices. While an advancement, these libraries presented a view of a somewhat complex state machine containing registers such as the current material properties and the current local or global transformation matrices. You had to drive this state machine, push register values onto a stack, change them, instruct the library to display a collection of triangles, and restore the registers at the right time.

The next major advancement was to further factor out common chores of graphics presentation into libraries that presented complex structured *models*, as exemplified in such systems as PHIGS, SGI's Inventor and Performer, VRML, and Microsoft's Direct3D RM (retained mode). The paradigm shift from presentation to modeling for geometry has had several practical benefits:

- *Ease of construction.* Models are generally easier for people to express and read than the corresponding presentation programs. (In the case of experienced programmers, there may be an initial period of *unlearning* presentation-oriented thinking habits, i.e., unconscious tendencies to think in terms of *how* to display some geometry, rather than simply what the geometry *is*.) In fact, model specifications are often not *programs* at all, but simply descriptions, such as "a red chair, doubled in size". Presentation specifications, on the other hand, generally are programs.
- *Authoring.* Content creation systems naturally construct models, because their end users think in terms of models, and typically have neither the expertise nor interest in programming model presentation.
- *Composability.* Models tend to be more robustly composable than presentation programs, thanks to the absence of side effects, which could otherwise interfere in subtle ways with the meaning of other components. Composability is a crucial factor in the scalability of any programming or modeling system, as well as the key to enabling powerful end-user features like cut-and-paste and drag-and-drop. The keys to robust composability are that (a) composition must construct the same kind of thing as the composed components, so that the result can be composed again, arbitrarily, and (b) composition operations do not allow interference among components. Note that there is an industry that sells a variety of specialized geometric models, but there is not one that sells specialized presentation code snippets.
- *Optimizability.* Model-based systems contain a *presentation sub-system* that contains code to render any model that can be constructed with the system. Because higher level information is available to the presentation sub-system than with presentation programs, there are many more opportunities for optimization. Examples include hierarchical culling, display-sensitive triangle generation from curved surfaces, set-up for various hidden surface removal algorithms when lacking Z-buffered hardware, and vertex data conversion from application representation to device representation. SGI's Performer and Microsoft's Direct3D RM products were largely motivated by these opportunities for optimization. Imagine how hard it would be to do these optimizations if the application explicitly managed each step of geometry presentation. It would be akin to reverse engineering the model out of the imperative presentation code.
- *Economy of scale.* Because the presentation sub-system is used for many different applications, it is worthwhile to invest considerably in optimization and functionality. When an application does its own presentation, such an investment is not as likely to be warranted.
- *Usefulness and longevity.* Models have broader usefulness and a longer lifetime than presentation programs, because models are platform independent. Presentation sub-systems can be separately tuned or totally re-implemented to run on a variety of radically different hardware architectures, from no graphics hardware, to SMP platforms, to SGI-like 3D hardware, and well beyond. Models will not only be *able* to be presented on these different architectures, but their presentation can exploit the best features of each architecture. Again, economy of scale makes this tuning and re-implementation work worthwhile.
- *Regulation.* The presentation sub-system can perform automatic level-of-detail management, determining the sequence of low-level presentation instructions executed dynamically, based on scene complexity, machine speed and load, etc. In contrast, a presentation-oriented application either hardwires a level-of-detail, and so is appropriate for only a narrow range of machines and circumstances, or must make a considerable investment in doing explicit, specialized regulation.

In spite of the benefits listed above, not everyone has made the shift from presentation to modeling of geometry. The primary source of resistance to this paradigm shift has been that it entails a loss of low level control of execution, and hence efficiency. As mentioned above, handing over low level execution control from the application to the presentation sub-system actually benefits execution efficiency where authors lack the significant resources and expertise required implement, optimize, and port their programs for all required platforms. In other cases, as in the case of current state-of-the-art commercial video games, the resources and expertise are available and well worth the considerable investment. An example is Doom, which would have been a failure at the time if implemented on top of a general-purpose presentation library. On the other hand, even Doom and its successors really adopt the modeling paradigm, in that they consist of a rendering engine paired with a modeling representation. In addition to the loss of direct control of efficiency, modeling tends to eliminate some flexibility in the form of presentation-level tricks that do not correspond to any expressible model. In our experience, these tricks tend not to scale well and are not composable, and in cases that do, are achievable through model extensibility.

There have been many other similar paradigm shifts, generally embodied in specialized languages sometimes with corresponding tools that generate the language. Examples include dialog box languages and editors; grammar languages and parser generators; page layout languages and desktop publishing programs; and high-level programming languages and compilers.

4 Modeling vs. presentation for animation

The conventional approach to constructing richly interactive animated content much like the old days of graphics rendering, as described briefly above, that is one must write sequential, imperative programs. (Much animation is in fact modeled rather than programmed, because it comes from animation authoring tools, but interaction is severely limited, for instance to hyper-linking.) These programs must explicitly manage common implementation chores that have nothing to do with the content of animation itself, but rather its *presentation* on a digital computer. These implementation chores include:

- sampling in time for simulation and frame generation, even though the animation is conceptually continuous;
- capturing and handling of sequences of motion input "events", even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel;
- management of network connections and remote messaging for distributed applications such as shared virtual spaces, multi-player games, and collaborative design, even though the various users and objects are conceptually in a single world;

The essence of modeled animation is to carry the presentation/modeling paradigm shift beyond static (non-time-varying) 3D geometry, and thus more broadly reap the kind of benefits described in the previous section. The extensions to static geometric modeling embodied in modeled animation include the following:

- Apply the modeling principle to richly model sound and 2D imagery. These types are as complex and important in their own right as geometry, and so are supported on equal footing with 3D, rather than as decorations on an essentially 3D representation, such as VRML's scene graphs. (In fact, for today's PC capabilities, sound and 2D imagery are more important than 3D geometry.) Going even further, treat the multitude of other data types that arise from 3D, 2D, and sonic modeling (transforms, points, colors, etc.) on an equal footing as well.
- Go beyond modeling of static geometry, images, etc., to behaviors and interaction—what one might call *temporal modeling*.
- Recognize that for a modeling representation to be sufficiently rich, it must inevitably be a quite expressive language, though not necessarily an imperative programming language. Even static modeling representations like VRML (Pesce [1995]) had to incorporate the essential mechanisms of a language, which are composition (through hierarchy and aggregation), naming, and information passing (through attributes), though in ad hoc, limited forms.

By extending modeling from static 3D to other types and to animation, we also extend the modeling benefits listed in the previous section. Most of these benefits translate in straightforward ways, but some possible non-obvious extensions are as follows:

- *Composability.* Temporal models compose into new temporal models, to an arbitrary level of complexity. There are no execution side-effects to cause interference among the composed components, nor are there any evaluation order dependencies. For example, in the examples above, consider the use of the relatively simple `spinPot` to help define the more complex `potSpin2`, and the use of `selectLeftRight` to define `bSign` and `growHowTo`.
- *Optimizability.* Techniques such as culling via spatial bounding volume hierarchies can be applied infrequently to temporal models, rather than at every frame on static models. Such analyzability is especially important for intensive computations like collision detection, which has been shown to be amenable to temporal analysis techniques. Moreover, because animation is modeled explicitly, rather than being the result of side-effects to a mutable static model carried out by

imperative code, the engine knows exactly what values and relationships are fixed and which ones vary dynamically. Note that it is exactly this knowledge that has proved vital to the success of programming language compilers. Most programmers gave up the control afforded by writing self-modifying code, and as a result, compilers gained enough information about the run-time behavior of a program to be able to perform significant optimization. As a result, most portions of even performance- and space-sensitive code are now written in languages like C or C++, rather than assembler. Many of the benefits of, and the objections to, modeling vs. presentation listed above directly apply to the issue of programming in C vs. assembly language.

- *Usefulness and longevity*. Because model definitions have no artificial sequentiality, temporal models may be executed in parallel, where parallel hardware is available. In contrast, imperative programs are notoriously difficult to parallelize and in practice must be rewritten.
- *Regulation*. The presentation of an interactive animation involves a multitude of sampling rates, including simulation parameter sampling, input sampling, geometry generation, geometry rendering, and image display. These many sampling rates may all be varied automatically, based on the computational and visual complexity of a scene, machine speed and load, etc. Moreover, computation of simulation parameters based on kinematics or dynamics can choose and adaptively vary numerical integration algorithms.

5 Language considerations

So far, we have used the term "modeling language" loosely. In this section, we make a more precise examination of the different possible notions of "language" and some of their pros and cons for practical use.

A language may be thought of as the combination of two complementary aspects. One aspect is domain-generic, and contains fundamental syntactic and semantic notions like definition and use of names for values and types, construction and application of functions or procedures, control flow, and typing rules. The other language aspect is a domain-specific vocabulary, describing, e.g., math operations on floating point numbers, string manipulation, lists and trees, and in our context, geometry, imagery, sound and animation.

Holding these two language aspects in mind, there are two strategies we could adopt in making concrete the idea of an animation modeling language, or any DSL, which we will call "integrated" and "embedded" respectively. In the integrated approach, the DSL combines both language aspects. In the embedded approach, the domain-specific vocabulary is introduced into an existing "host" programming language. While these two strategies may be similar in spirit, the pragmatics of carrying them out differ considerably.

The chief advantage of integration is that one can have a perfectly suited language, semantically and syntactically, while the embedded approach requires toleration of compromises made to accommodate a broad range of domains. In return for this toleration, the embedded DSL approach allows us to use already existing language infrastructure.

To be useful in practice, not just a toy or a research experiment, a complete DSL needs several components, well designed and well executed:

- A language definition. Despite the best intentions of their original designers, successful DSL's (ones with users) tend to grow, eventually incorporating more and more general purpose language features. When this growth is not anticipated, the results can be ugly.
- A language implementation. Depending on the domain, an interpreter may suffice, but for some cases, including real-time animation, compilation is important. A good compiler, such as the Glasgow Haskell Compiler (Peyton Jones and Santos [1996]), requires years to develop.
- Environment tools. Programmers need debuggers and profilers to get their programs working correctly and efficiently. Also, inherent in domain-specificity, there needs to be a way to package up components of functionality in such a way that it can inter-operate with components implemented in other languages, domain-specific or otherwise.
- Educational material. Users must be provided with tutorials to get them started, and reference manuals to fill in the details.

Given this list, we have ample incentive to try to make the embedded DSL approach work, if we can find a sufficiently suitable existing host language. We now take a closer look at the question of what features constitute suitability.

6 Choosing a host language for modeled animation

We have found a variety of host language features to be helpful for animation modeling, while others were harmful. The helpful features include the following, some of these features are obvious from a programming language perspective, but are in fact missing or very weakly present in popular model formats for geometry and animation.

- *Expressions.* Models are specified primarily in terms of other models, applying various kinds of transformations, forming aggregates, transforming some more, etc. Expressions, in the programming language sense, are well suited for this *compositional* style of specification, since they nest conveniently and suggest manipulation of *values* (models) rather than *effects* (presentations). One kind of expression that is particularly useful is the conditional, as in C's often ignored: "*cond ? exp1 : exp2*".
- *Definition.* In order to use a model more than once, or to separate the definition of a model from its uses, there needs to be a mechanism for referring to a single model any number of times in different contexts. A simple and general such mechanism is the definition of names for models denoted by expressions, together with the use of names to denote the corresponding models. Such definitions should have controllable scope, such as introduced by "where" in the some of the examples above.
- *General parameterization.* Values such as numbers, strings and are not nearly as interesting a set of reusable building blocks as are the *functions* that create these values. Exactly the same is true for values/models such as geometry, images, sounds, transformations, and animations. The really powerfully reusable building blocks tend to be parameterized models, such as `spinPot` and `leftRightSelect` above, and therefore a modeling language needs a mechanism for expressing functions from arguments of arbitrary types to results of arbitrary types.
- *Higher-order programming (first class functions).* Higher-order functions allow succinct expression an encapsulation of useful domain-specific programming patterns. Consequently, it is useful to allow for parameterized models to accept other *parameterized* models as arguments and/or produce them as results. As a particular example, response to user interaction events is often expressed in terms of call-backs. In a higher-order language, these call-backs may be specified succinctly, using lambda-abstraction or locally-defined functions. (Strong static typing eliminates the need for unsafe type coercion or run-time checking.) In the examples above, `withSpin` makes critical use of higher-order programming.
- *Strong, static typing.* Models and their components are of a variety of different types, such as geometry, image, sound, 2D and 3D transform, 2D and 3D point and vector, color, number and Boolean, as well as animations over all of these types, and events yielding information of all of these types. A static type system guides authors toward meaningful model descriptions, enabling helpful error messages before execution. Static typing also improves performance by eliminating the need for run-time type checking, while retaining execution safety. In order not to clutter a model definition, it is helpful is types can be inferred automatically, rather than always being specified explicitly.
- *Parametric polymorphism.* Animation is a polymorphic concept, applying to geometry, images, sound, 2D and 3D points vectors, colors, numbers etc. Similarly, reactive animation makes essential use of the polymorphic notion of an *event* occurrences of which carry with them not only a time, but also a value of some type. Several of Fran's animation- and event-building operations, such as "`untilB`", "`==>`" and "`.|.`", apply to an infinite family of types. Note that non-polymorphic languages generally have polymorphic primitives, such as conditionals. To serve as a host language for an embedded DSL, however, the polymorphism must be available for the embedded vocabulary, as in the function `selectLeftRight`, which was applied to numbers and to strings above.
- *Notational flexibility.* It is convenient to give new, domain-specific, meanings to old names. In particular, Fran overloads most of the names of the standard math functions, e.g., "`sin`" and "+", to operate at the level of animations. We even overload constants, e.g., "`pi`" and "1", to denote animations (though not very animated ones). We also introduce new infix operators with suitable associativity and binding strength. While "merely" a notational convenience, this notational flexibility is largely responsible for giving the "look and feel" of a tailored domain specific language, and makes the resulting programs much easier to read than they would be if we had to introduce a whole new collection of non-infix names.
- *Automatic garbage collection.* An animation

typically contains many components that contribute for a short while, or in any case, less than the full duration of the animation. Automatic garbage collection makes for safe and efficient memory use.

- *Laziness*. An interactive animation is a "big" value, often infinitely big, containing repetition and branching. It is important, even crucial, that parts of an animation be available for consumption before the rest of the animation has actually been produced. The idea of laziness is to postpone production of parts of a value until the last possible moment, i.e., when those parts need to be consumed for display. Often parts are completely unused, and so should never actually be computed. For example, in a computer game, many possible branches are not taken and many simulated characters are not seen during the play of a single game. As a simpler example, the animations produced by `bSign` and `growHowTo` can have an infinite number of phase changes, according to user input, but they available immediately for partial consumption.

  What are usually thought of as primitive control structures, such as conditionals and iteration, are often definable in lazy languages. As a consequence, "domain-specific control structures" are also definable. (Higher-order programming with lambda abstraction makes it possible to define domain-specific variable binding constructs as well.) For instance, one could define animation repetition operators in Fran.

  Laziness also plays a role complementary to garbage collection, for efficient use of memory. Laziness delays consumption of memory until just before an animation component is needed, while garbage collection frees the memory when an animation component is no longer needed.

- *Modules*. Like conventional programs, model specifications can grow to be quite complex, and so should be specifiable in parts by different authors and in different files distributed throughout the Internet. Moreover, it should be possible to compile these modules into an executable form such as Intel binary or Java byte-code, with formal interfaces that state the names and types of values and functions implemented in the module.

Imperative programming languages, such as C, C++, Java and Visual Basic, have *statements* in addition to *expressions*, and in fact, emphasize statements over expressions. For example, in these languages, it is possible to introduce a scoped variable in a statement, but not in an expression. Also, **if** works on statements, though C has its ternary **?:** expression operator. While expressions are primarily for denoting values, statements are for denoting changes to an internal or external state. State changes certainly occur during *presentation* of a model, but are not appropriate in the model itself, as they interfere with composability, optimizability, and multithreaded, parallel and distributed execution. Common language features that are statement-oriented, and which thus do *not* useful for modeled animation, include the following:

- *Sequencing*. Without statements, there is no role for the usual notion of sequencing, which is executing multiple statements in serial, and relies on *implicit communication* of information from one statement to the next through side effects.
- Conditional statements.
- *Sequential iteration*. Really just a compact way to specify possibly infinite sequencing and conditional execution.

Given the language requirements and non-requirements above, we now return to the "integrated-vs-embedded" question, keeping in mind that design and implementation of a new programming language and development tools, and creation of required educational material are formidable tasks, not to be undertaken unless genuinely necessary. Fortunately, there are well-suited existing languages, the so-called "statically typed, higher-order, purely functional" languages. Of those languages, *Haskell* (Hudak et al [1992b], Hudak and Fasel [1992a]) has the largest following, has an international standard (Haskell 1.4) and is undergoing considerable development. For these reasons, we have chosen Haskell for our own implementation of the ideals of modeled animation. Other languages can be used as well, with varying tradeoffs. For example, Java is more popular than Haskell, and while predominately statement-oriented, it does support garbage collection.

While neither the current development tools and educational material for Haskell programming, nor the size of the Haskell programming community, is impressive compared to those of mainstream languages, we believe that both are sufficient to act as a seed, with which to generate initial compelling applications. We hope that these initial applications will inspire curiosity and creativity of a somewhat larger set of programmers, leading to better development tools and written materials, yet more compelling applications, and so on, in a positive feedback cycle.

Aside from issues of familiarity, there will always be an important role for imperative computation in the construction of *complete* applications, which is best described using statement-oriented programming languages. One then could throw such features into a modeling language, or even try to force imperative programming languages to also serve as modeling languages. We prefer the approach of *multi-lingual integration*, which is to support construction of application modules in a variety of languages and then combine the parts, generally in compiled form, with a language neutral tool.

7 Related work

The idea of an "domain-specific embedded language" is, we believe, the central message in Landin's seminal "700" paper:

> Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages. [...] ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. (Landin [1966]).

Arya [1994] used a lazy functional language to model non-interactive 2D animation as lazy lists of pictures, constructed using list combinators. This work was the original inspiration for our own; we have extended it to interactivity, continuous time, and many other types besides images.

TBAG modeled animations over various types as functions over continuous time (Elliott et al [1994], Schechter et al [1994]). It also used the idea of lifting function on static values into functions on animations, which we adopted for Fran. Unlike Fran, however, reactivity was handled imperatively, through constraint assertion and retraction, performed by an application program. Like Fran, TBAG was an embedded language, but it used C++ as its host language, in an attempt to appeal to a wider audience. The C++ template facility was adequate for parametric polymorphism. The notation was in some ways even more malleable than in Haskell, because C++ overloading is *genuinely* ad hoc. On the other hand, unlike Haskell, C++ only admits a small fixed set of infix operators. The greatest failings of C++ (or Java) as a host language for a modeling language are its lack of an expression-level "let", and the absence of higher-order functions. The latter may be simulated with objects, but without a notational equivalent to lambda expressions.

Obliq-3D is another 3D animation system embedded in a more general purpose programming language (Najork and Brown [1995]). However, its host language is primarily imperative and object-oriented, rather than functional. Accordingly, Obliq-3D's models are initially constructed, and then modified, by means of side-effects. In this way it is reminiscent of Inventor (Strauss [1993]).

Direct Animation is a library developed at Microsoft to support interactive animation (Microsoft [1997]). It is designed to be used from mainstream imperative languages such as Java, and mixes the functional and imperative approaches. Fran and Direct Animation both grew out of an earlier design called *ActiveVRML* (Elliott [1996]), which was an "integrated" DSL.

There are also several languages designed around a *synchronous data-flow* notion of computation, including Signal (Gautier et al [1987]) and Lustre (Caspi et al [1987]), which were specifically designed for control of real-time systems. In Signal, the most fundamental idea is that of a *signal*, a time-ordered sequence of values. Unlike Fran, however, time is not a value, but rather is implicit in the ordering of values in a signal. By its very nature time is thus discrete rather than continuous, with emphasis on the relative ordering of values in a data-flow-like framework. The designers of Signal have also developed a clock calculus with which one can reason about Signal programs. Lustre is a language similar to Signal, rooted again in the notion of a sequence, and owing much of its nature to Lucid (Wadge and Ashcroft [1985]).

8 Conclusions

Traditionally the programming of interactive 3D and multimedia animations has been a complex and tedious task. We have argued that one source of difficulty is that the languages used are suited to describe how to present animations, and in such descriptions the essential nature of an animation, i.e., what an animation is, becomes lost in details of how to present it. Focusing on the "what" of animation, i.e., modeling, rather than the "how" of its presentation, yields a much simpler and more composable programming style. The modeling approach requires a new language, but this new language can be synthesized by adding a domain-dependent vocabulary to an existing domain-independent host language. We have found Haskell quite well-suited, as demonstrated in a collection of sample animation definitions.

A running theme of this paper has been economy of scale. We recommend making choices that amortize effort required over several uses of the fruits of that effort. The alternatives are poor quality or impractically high cost. Specifically:

- "Modeling" vs "presentation". Graphics modeling allows reuse of a single graphics presentation engine, and temporal modeling allows reuse of a single temporal presentation engine.
- "Embedded" vs "integrated" language. Languages, if they are to be genuinely useful, require a large investment of effort. An embedded language inherits design, compilers, environment tools, and educational material from its host language.
- Composability. Because modeled, parameterized animations are neatly composable, they may be reused in a variety contexts, instead of being repeatedly reinvented with slight variations for each similar situation.

A notable exception to the necessity of modeling, embedding and composability for high quality interactive animation is in software that can sell in huge quantity, which then exploits an end-user economy of scale. The unfortunate consequence to this exception, however, is a kind of mainstreaming of the content, as in violent video games. Fortunately, however, even these games are often implemented using the modeling approach, and allow consumers to create new characters and worlds for them.

There are ample opportunities for future work in modeled animation, including the following.

- Multi-lingual integration. We believe that in order for Haskell, or any other non-mainstream language to make a serious contribution in the software industry, it should be cast not as a language for implementing entire *applications*, but rather *software components*. This identity then implies strong support for generating language-independent calling interfaces. As a concrete goal, one should be able to program animation modules in Haskell, compile them into binaries with COM interfaces, and then distribute them. A Java or Visual Basic programmer should then be able to wire together the Haskell-based animation components without knowing in what language they were implemented.
- Domain-specific optimization. In theory, it is possible for a domain-generic compiler to do domain-specific compilation, by using various forms of partial evaluation. We intend to investigate this approach, by using the Glasgow Haskell compiler (Peyton Jones and Santos [1996]), perhaps with some domain-generic enhancements.
- Notational compromises. As mentioned above, using Haskell required only a few compromises. One has to do with overloading. We cannot, for instance, use "`+`" for the addition of 2D or 3D points and vectors (or even "`.+^`", which now can be used for 2D or 3D, but not both). Similarly, we cannot use "`==`" for the lifted form of equality, applying to two like-typed animations to yield a boolean animation. Extending Haskell to allow "multi-parameter type classes" might eliminate some of these compromises.

9 Acknowledgements

My thoughts on "domain-specific embedded language" have been greatly influenced by Paul Hudak. Philip Wadler pointed out the connection to Landin's "700" paper. Todd Knoblock and Jim Kajiya helped to explore the basic ideas of modeled animation. Sigbjorn Finne helped with the implementation during a summer research internship. Alastair Reid made many implementation improvements. Paul Hudak, Alastair Reid, and John Peterson at Yale provided many helpful discussions about functional animation, how to use Haskell well, and lazy functional programming in general. Gary Shu Ling helped get Fran running under GHC. Byron Cook gave many helpful comments on an earlier draft to improve readability.

10 Availability

Fran runs under Windows 95 and NT 4.0, and is freely available at http://www.research.microsoft.com/~conal/Fran/.

References

Kavi Arya [January 1994], "A Functional Animation Starter-Kit", *Journal of Functional Programming*, 4(1):1-18.

P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice [January 1987], "Lustre: A Declarative Language for Programming Synchronous Systems", in *14th ACM Symposium. on Principles of Programming Languages*.

Anthony Daniels [1997], "Fran in Action!", in preparation, http://www.cs.nott.ac.uk/~acd/action.ps

Conal Elliott [February 1996], "A Brief Introduction to ActiveVRML", Technical Report MSR-TR-96-05, Microsoft Research, ftp://ftp.research.microsoft.com/pub/tr/tr-96-05.ps

Conal Elliott [1997], "Composing Reactive Animations", To appear in *Dr. Dobb's Journal*, http://www.research.microsoft.com/~conal/fran/tutorial.htm .

Conal Elliott, Greg Schechter, Ricky Yeung and Salim Abi-Ezzi [July 1994], "TBAG: a High Level Framework for Interactive, Animated 3D Graphics Applications", in Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida)*, pages 421-434. ACM Press, http://www.research.microsoft.com/~conal/tbag/papers/siggraph94.ps

Conal Elliott and Paul Hudak [June 1997], "Functional Reactive Animation", in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, http://www.research.microsoft.com/~conal/papers/icfp97.ps

Thierry Gautier, Paul Le Guernic, and Loic Besnard [1987], "Signal: A Declarative Language for Synchronous Programming of Real-Time Systems", in Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, edited by G. Goos and J. Hartmanis, pages 257-277. Springer-Verlag, 1987.

Paul Hudak and Joseph H. Fasel [May 1992a], "A Gentle Introduction to Haskell". *SIGPLAN Notices*, 27(5). See http://haskell.org/tutorial/index.html for latest version.

Paul Hudak and Simon L. Peyton Jones and Philip Wadler (editors) [March 1992b], "Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)", *SIGPLAN Notices*. See http://haskell.org/report/index.html for latest version.

Peter. J. Landin [March 1966], "The Next 700 Programming Languages", Communications of the ACM, 9(3), pp. 157-164.

Microsoft [1997], DirectAnimation, in the Microsoft DirectX web page, http://www.microsoft.com/directx.

Marc A. Najork and Marc H. Brown [June 1995], "Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System", IEEE Transaction on Visualization and Computer Graphics, 1(2).

Mark Pesce [1995], VRML Browsing and Building Cyberspace: the Definitive Resource for VRML Technology, New Riders Publishing.

John Peterson and Gary Shu Ling [1997], "Fran User's Manual", http://www.haskell.org/fran/fran.html

Simon Peyton Jones and Andre Santos [1996], "Compiling Haskell by Program Transformation: a Report from the Trenches", ESOP '96: 6th European Symposium on Programming, Linköping Sweden, April 22—24, 1996, Lecture Notes in Computer Science, Vol. 1058, Springer-Verlag Inc. http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/comp-by-trans.ps.gz

Greg Schechter, Conal Elliott, Ricky Yeung and Salim Abi-Ezzi [1994], "Functional 3D Graphics in C++ - with an Object-Oriented, Multiple Dispatching Implementation", in *Proceedings of the 1994 Eurographics Object-Oriented Graphics Workshop*. Springer Verlag, http://www.research.microsoft.com/~conal/papers/eoog94.ps

Paul S. Strauss [October 1993], "IRIS Inventor, A 3D Graphics Toolkit", in Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications, pp. 192-200.

W.W. Wadge and E.A. Ashcroft [1985], *Lucid, the Dataflow Programming Language*. Academic Press U.K..

A Special-Purpose Language for Picture-Drawing

Samuel N. Kamin* David Hyatt†
Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
{s-kamin, d-hyatt}@uiuc.edu

Abstract

Special purpose languages are typically characterized by a type of primitive data and domain-specific operations on this data. One approach to special purpose language design is to embed the data and operations of the language within an existing functional language. The data can be defined using the type constructions provided by the functional language, and the special purpose language then inherits all of the features of the more general language. In this paper we outline a domain-specific language, FPIC, for the representation of two-dimensional pictures. The primitive data and operations are defined in ML. We outline the operations provided by the language, illustrate the power of the language with examples, and discuss the design process.

1 Introduction

FPIC is a special-purpose language for drawing simple pictures. It was built by defining types and functions in the functional language Standard ML [6]. This method of construction is easy and results in a language with many useful features. In addition to being concise for small examples, FPIC is powerful enough to allow the programming of large programs and program libraries, an area in which many special-purpose languages are weak.

Functional programming has been characterized in many ways. Our view is that it represents an approach to language design. This approach holds that some mathematical constructs—products, functions, disjoint unions, and others—are fundamental in computing and should be well supported in programming languages. This support means allowing the creation of "first-class" values of each type, that is, values not subject to arbitrary restrictions based on the type. It also means providing operations appropriate to those types in a concise, non-bureaucratic form.

In our view, this approach to language design is perfectly suited to the design of special-purpose languages. These languages are usually characterized by a type of primitive data specific to a problem domain, and operations on those data. These data can be incorporated into a language having the type constructions just mentioned. In fact, they can be incorporated into an *existing* functional language; the type constructions will apply to the new data, and the entire language will then become a special-purpose language, with its many other features included "for free."

The principal weakness of many special-purpose languages is that, beyond a concise and natural syntax, and efficient implementation, for the values and operations specific to the domain, overarching language structure is weak. This weakness would be very significantly mitigated if special-purpose languages were routinely designed—or at least prototyped—in the way we have outlined.

This paper is a case study of the design of FPIC according to this philosophy. We describe the process by which we decided what the primitive data were and how they should behave, then describe the language itself with numerous examples. Our emphasis throughout is on the advantages obtained by having the functional language superstructure of Standard ML as part of FPIC.

*Partially supported by NSF Grant CCR 96–19655.

†Current address: Netscape Communications Corp., Mountain View, CA, hyatt@netscape.com.

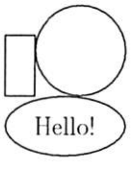
2 Simple FPIC Examples

PIC [4] is a language for drawing simple pictures, such as trees and block diagrams. It has primitives for drawing boxes, circles, and other shapes, with or without labels, and for drawing lines and arrows between them. It also has a facility for naming points on pictures, to be used, for example, as the endpoints of lines and arrows. These constructs are provided in a concise syntax, with a simple language structure (including loops) added on.

FPIC was inspired by PIC. Our goal was to demonstrate that we could benefit by following the language design philosophy outlined in the introduction. That is, by using essentially the same primitive data types and operations as in PIC, but embedding them in a functional language, we could obtain a far more powerful language than PIC and do so at a far lower cost than if we had built the language from scratch.

In this section, we present a few examples to show the principal *primitive* operations of FPIC, making only minimal use of the programming features of Standard ML. Section 5 gives many more examples, emphasizing the utility of the features of ML in combination with the FPIC primitives.

The most basic primitives are those for drawing simple shapes and placing them next to one another:[1]

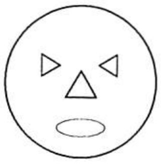


```
box 1.0 2.0 hseq circle 1.5 vseq
     label "\\Huge Hello!" (oval 2.0 1.0);
```

hseq and vseq represent the operations of placing pictures next to one another, either horizontally or vertically. (Note that backslashes inside quotes must be doubled.)

Pictures can be moved and otherwise transformed in various ways. In this example, we use ML's name definition facility in the first line. dtriangle is a "default triangle;" similarly for doval (and dcircle and dbox later in the paper).

[1] Appendix A gives a concise overview of ML syntax. Appendix B lists all the FPIC primitives used in the examples in this paper, indicating the types of their arguments, whether or not they are infix, and what they return.



```
val nose = dtriangle;
circle 2.5
  seq (nose at (2.0,2.0))
  seq (nose rotate ~90.0 scale 0.7 at (1.2,2.7))
  seq (nose rotate 90.0 scale 0.7 at (3.1,2.7))
  seq doval scaleXY (0.5,0.3) at (1.7,0.7);
```

The seq operation simply places pictures on top of one another, without moving them either right or down. The expression *pic* at *point* draws the picture *pic* with its reference point (the lower-left corner) at *point*.

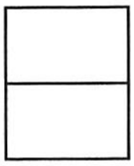
An important feature of PIC, which we have adopted in FPIC, is the ability to name points in a picture and subsequently refer to them. The compass points—s for south, ne for northeast, and so on, plus c for center—are automatically defined for every picture. This allows us to eliminate some of the guesswork in the previous example:

```
val face = circle 2.5;
val facecenter = face pt "c";
val lefteye = nose rotate ~90.0 scale 0.7;
val righteye = nose rotate 90.0 scale 0.7;
val mouth = doval scaleXY (0.5,0.3);
face seq (nose centeredAt facecenter)
     seq (lefteye centeredAt (facecenter -- (1.0,~0.7)))
     seq (righteye centeredAt (facecenter ++ (1.0,0.7)))
     seq (mouth centeredAt (facecenter -- (0.0,1.5)));
```

Named points can be added to a picture. A third way to produce the same "pumpkin face" is to draw the face and name the locations of the facial features:

```
val face =
    let val f = circle 2.5
        val facecenter = f pt "c"
    in namePts f
        [("nosepos", facecenter),
         ("lefteyepos", facecenter -- (0.9,~0.8)),
         ("righteyepos", facecenter ++ (0.9,0.8)),
         ("mouthpos", facecenter -- (0.0,1.5))]
    end;
face seq (nose centeredAt (face pt "nosepos"))
     seq (lefteye centeredAt (face pt "lefteyepos"))
     seq (righteye centeredAt (face pt "righteyepos"))
     seq (mouth centeredAt (face pt "mouthpos"));
```

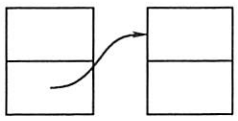
Pictures can be named as well as points. This is useful when a number of points on a picture may be of interest. For example, `cell` is a "cons cell" consisting of two boxes vertically stacked; for future reference, the individual boxes are named car and cdr, respectively:



```
val cell = namePic dbox "car"
           vseq namePic dbox "cdr";
```

In addition to optionally being named, subpictures are automatically numbered. *p* `nthpic` *i* is the *i*th subpicture in *p*.

Finally, another important feature of PIC and FPIC is the ability to draw lines and arrows. In this picture, two cells are drawn with a curved arrow from the cdr of the first to the car of the second:



```
let val cells = (namePic cell "left") hseq
                (hspace 1.0) hseq
                (namePic cell "right")
    val source = cells pic "left" pic "cdr" pt "c"
    val target = cells pic "right" pic "car" pt "w"
in cells seq (bezier source
                     (source ++ (1.0,0.0))
                     (target -- (1.0,0.0))
                     target
             withArrowStyle "->")
end;
```

3 How to design a special-purpose language

Much as we try not to, we often design a new language by thinking in terms of the syntax we would like it to have. For special-purpose languages, this generally means concentrating on the syntax of the domain-specific data and operations.

In our view, thinking about the desired concrete syntax of the new language is not at all a bad place to begin the design process, as long as that step is understood in its proper relation to the overall language design. In designing FPIC, we had an existing language, PIC, to start from. We used the syntax of PIC to help answer what we consider the most crucial question in the design process: what does each piece of syntax *mean*? In ordinary programming terms, syntactic phrases have some intuitive operational meaning; "`circle 2.0`" means "draw a circle with radius 2.0." A lesson from denotational semantics is that phrases can be given precise meanings as values of well-defined sets. It is this notion of "meaning" that we found useful in designing FPIC. We wish to give the reader some idea of the thought process we went through.

We will call the set in which the meaning of "`circle 2.0`" resides *Picture*. The question is, what exactly is in this set? What is a picture? The most obvious answer is that it is some concrete representation of a picture, for example, an array of pixels or a list of drawing commands. The precise representation does not matter to us, so we will just give the representation the generic name *BitMap*. So, our first idea is to say

$$\mathit{Picture} = \mathit{BitMap}$$

However, a closer look at PIC tells us that this can't be quite right. In PIC, one can write, for example,

```
box; box
```

meaning to draw one box next to another. (We would write this as `dbox hseq dbox`.) If each box is a *Picture*, and a *Picture* is just a bitmap, then these two boxes would represent the *same* bitmap, which would mean precisely the same pixels drawn at the same location!

We should instead say that a *Picture* is the *capability* to draw a bitmap, given a location at which to draw it; in other words, it is a *function* from locations to drawing commands:

$$\mathit{Picture} = \mathit{Location} \rightarrow \mathit{BitMap}$$

This is close, but we have also to deal with the picture that we write (from now on, we use FPIC syntax to avoid confusion):

```
dbox scale 5.0
```

Again, the bitmap cannot be determined from the value of `dbox`, even after knowing its location. We might try

$$Picture = Location \rightarrow ScaleFactor \rightarrow BitMap$$

which is basically correct, but we also need to deal with color, line width, and a host of other ways in which the simple bitmap might be altered.

From this point of view, the value of dbox is just the barest indication of what will actually show up on the printed page. We represent this by defining

$$Picture = GraphicsContext \rightarrow BitMap$$

where *GraphicsContext* contains information about any linear transformations that may have been applied to the picture, as well as color, fill style, line width, and so on.[2]

We're not quite done. We know that we can refer to the points on a picture, for example dbox pt "nw". Evidently, dbox means something more than a bitmap. It means, in addition, a set of named points, which we call an *Environment.* This brings us to the definition we actually use in FPIC:

$$Picture = (GraphicsContext \rightarrow BitMap) \times Environment$$

This definition—and it is by no means the only one possible—is the most important in the design of FPIC. Just as the domains in the denotational semantics of a language are chosen to match the properties of that language, so here the definition of *Picture* determines, more than any other single thing, the properties of FPIC. Indeed, once this definition is made—along with the precise definitions of *GraphicsContext* and *Environment*—the rest of the language largely falls out.

When embedding a language in a functional language like Standard ML, this type definition can also guide the implementation. Indeed, we have used exactly the type above, written in ML as

```
type Picture =
    (GraphicsContext->BitMap) * Environment;
```

as the type of pictures. Many of the basic operations on pictures, and their implementations, are suggested directly by this type definition.

[2]PostScript [1] has a similar notion of "graphics context."

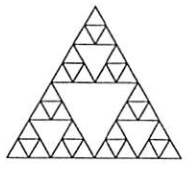
4 The FPIC User's Manual

Since FPIC includes Standard ML, the manual is either very long or very short, depending upon how you look at it. In any case, FPIC consists of about 160 functions, amounting to about 1200 lines of ML code. In Appendix B, we list all the FPIC primitives that are used in the examples in this paper.

5 FPIC Examples

The examples of this section show how the features of Standard ML, when combined with the primitives of FPIC, create a powerful language for constructing pictures.

In a functional language, some fancy drawings are relatively easy to do. For example, here is the "Sierpinski gasket" of order 3:



```
fun sierpinski 0 = dtriangle
  | sierpinski n =
      let val s = sierpinski (n-1) scaleWithPoint
                         ((0.5,0.5),(0.0,0.0))
      in s hseq s seq (s at ((width s)/2.0, height s))
      end;
sierpinski 3;
```

(*pic* scaleWithPoint (*s*, *point*) scales picture *pic* by a factor *s* while keeping *point* fixed.)

However, of more interest in practice is the ability to create reusable pieces of pictures to ease the programming burden. Here is where functional programming shines.

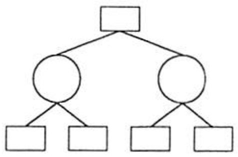
5.1 Defining lines

In FPIC, a line is simply a function from two points to a picture. Any drawing can be parameterized by a line to create a variety of effects.

Here is a simple function to draw trees. Its arguments are a picture representing the root of the tree and a list of pictures representing its children.

```
fun drawtree root subtrees =
let val bottom = hseqtopsplist 1.0 subtrees
    val top = placePt root "s"
              (bottom pt "n" ++ (0.0,1.0))
    val rootsouth = top pt "s"
in group (top seq bottom seq
    (seqlist
       (map (fn p => line rootsouth (p pt "n"))
       (pics bottom))))
end;
```

It draws a tree with its nodes connected by lines:

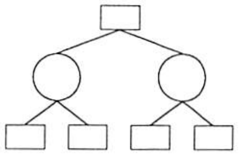


```
let val t = drawtree dcircle [dbox, dbox]
  in drawtree dbox [t, t] end;
```

We can define a function drawTreeWithArrow that would draw arrows instead of lines, simply by replacing "line" by "arrow." However, we can do better in ML, making the line-drawing function a parameter. drawTree becomes

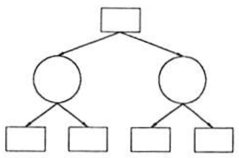
```
fun drawtree root subtrees linefun =
   ... exactly the same, until the end ...
    (seqlist
      (map (fn p => linefun rootsouth (p pt "n"))
      (pics bottom))))
end;
```

Then the tree above would be written as



```
let val t = drawtree dcircle [dbox, dbox] line
  in drawtree dbox [t, t] line end;
```

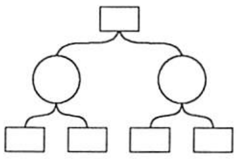
and we could also write



```
let val t = drawtree dcircle [dbox, dbox] arrow
  in drawtree dbox [t, t] arrow end;
```

Moreover, we can define our own line-drawing functions. We have seen earlier, in the cons-cell example, how to draw a curvy line. We use the same idea, except that here we want our curvy line to begin with a vertical leg rather than a horizontal one. Again, keep in mind that a line-drawing function is just a function from two points to a picture, nothing more or less:

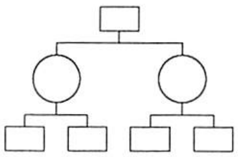
```
fun curvedvline pt1 pt2 =
   bezier pt1 (pt1--(0.0,1.0))
          (pt2++(0.0,1.0)) pt2;
```



```
let val t =
     drawtree dcircle [dbox, dbox] curvedvline
  in drawtree dbox [t, t] curvedvline end;
```

Two more examples are a line-drawing function that uses a "Manhattan geometry":

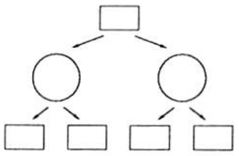
```
fun manline pt1 pt2 =
   let val ymid = (snd pt1 + snd pt2)/2.0
   in seqlist
        [line pt1 (fst pt1, ymid),
         line (fst pt1, ymid) (fst pt2, ymid),
         line (fst pt2, ymid) pt2]
   end;
```



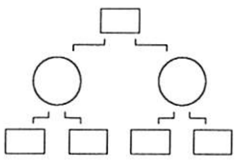
```
let val t = drawtree dcircle [dbox, dbox] manline
  in drawtree dbox [t, t] manline end;
```

and a function that draws short lines of whatever kind:

```
fun shortline linefun pt1 pt2 =
    let val diff = pt2 -- pt1;
        val pt1' = pt1 ++ diff**(0.25,0.25);
        val pt2' = pt2 -- diff**(0.25,0.25)
    in linefun pt1' pt2'
    end;
```



```
let val t = drawtree dcircle [dbox, dbox]
                              (shortline arrow)
  in drawtree dbox [t, t] (shortline arrow) end;
```



```
let val t = drawtree dcircle [dbox, dbox]
                              (shortline manline)
  in drawtree dbox [t, t] (shortline manline) end;
```

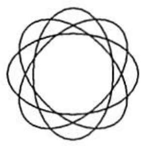
5.2 Defining sequencing operators

The functions that put pictures together into larger pictures are of key importance. In FPIC, these are generally binary operations with infix syntax. The basic sequencing operation is seq, which simply draws two pictures without prejudice; hseq and vseq, among others, combine seq with some translation of the second picture.

Again, the ability to define new sequencing operations is of the greatest interest. A sequencing operation is a function from a pair of pictures to a picture. The constructed picture should include the two pictures.

A simple example is cseq, which aligns the centers of two pictures:

```
infix 6 cseq;
fun (p1 cseq p2) = (p1, center p1) align
                   (p2, center p2);
```

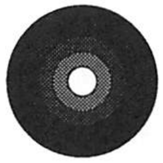


```
doval cseq (doval rotate 45.0)
      cseq (doval rotate 90.0)
      cseq (doval rotate 135.0);
```

Given a sequence operation *seq*, we often use the related operation *seq*list, which applies to lists of pictures. Specifically:

$$seq\texttt{list}\ [p_1, p_2, \ldots, p_n] \equiv p_1\ seq\ p_2\ seq \ldots seq\ p_n$$

The function mkseqlist creates the list version of a sequencing operation from the ordinary binary version.



```
val cseqlist = mkseqlist (op cseq);

val bullseye = cseqlist (map
      (fn rad => circle rad withFillColor
                    (1.0/rad, 1.0/rad, 1.0/rad))
    [5.0, 4.0, 3.0, 2.0, 1.0]);
```

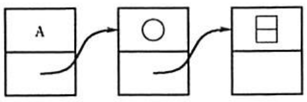
The function mkseqfun is provided to facilitate the creation of new sequencing functions. Given two functions f and g from pictures to points, it creates the sequencing operation that, given two pictures p and q, draws the two so that fp and gq coincide. For example, in the tree-drawing function presented earlier, we used the sequencing operation hseqtopsplist, the list version of the sequencing operation hseqtopsp. The latter sequences two pictures horizontally with their tops aligned, adding some space between them. It is not built-in, but is defined as follows:

```
fun hseqtopsp gap =
  mkseqfun (fn p => northeast (p right 1.0))
           northwest;
```

The cons-cell example suggests another kind of sequencing: sequencing with an arrow. cellseq has as its arguments two cons cells—that is, two pictures that are presumed to have subpictures called cdr and car, respectively—and draws them a bit separated, with a curvy arrow. To allow more than one cell to be sequenced in this way, the combination of cells is defined to have car and cdr subpictures itself.

```
infix 7 cellseq;
fun cell1 cellseq cell2 =
   let val c = (group cell1)
               hseq (hspace 1.0)
               hseq (group cell2)
       val cells = c seq curvedharrow
                   (c nthpic 1 pic "cdr" pt "c")
                   (c nthpic 3 pic "car" pt "w")
   in addNamedPics cells
            [("car", cells nthpic 1 pic "car"),
             ("cdr", cells nthpic 3 pic "cdr")]
   end;
```

For this example, we have defined labelledCell, which draws a cons cell with a label in its car:

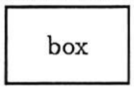


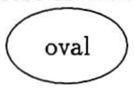
```
fun labelledCell L =
   cell seq (L centeredAt (cell pic "car" pt "c"));

labelledCell (text "A")
  cellseq
labelledCell (dcircle scaleTo (0.5, 0.5))
  cellseq
labelledCell (cell scale 0.3);
```

6 Latex integration

FPIC pictures are included in LaTeX documents by using the fpic command:

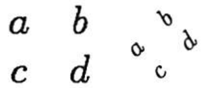
```
\fpic{picture-name}{
    ... FPIC specification ...
}
```

The specified picture becomes an ordinary box in LaTeX, so that it can be included anywhere within the document like any other piece of text. For example, this 

and this 

are placed in-line. Notice that LaTeX knows their sizes and creates the right amount of horizontal and vertical space for them.

The other side of this coin is the inclusion of TeX text within FPIC pictures. The text function turns a string—interpreted as LaTeX input—into an FPIC picture. Any LaTeX input can be used here, and once the text function is applied it becomes subject to the same transformations as any other piece of text:

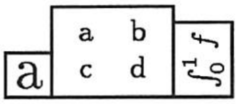


```
let val A = text
  "$\\begin{array}{cc} a & b \\\\ c & d \\end{array}$"
in hseqsplist 0.5 [A scale 1.5, A rotate 45.0]
end;
```

The only difficulty here is that FPIC does not know how large the text will turn out to be. We adopt a solution to this problem similar, in essence, to that used by Chailloux and Suarez in *mlPicTeX* [2]. When FPIC is first run, it produces LaTeX code that causes LaTeX to write the size of the text to its intermediate (aux) file; on subsequent runs, FPIC reads this information from that file. As is common with LaTeX utilities, this requires that LaTeX be run twice when a new picture with text is added, or when the text of an existing picture is changed, to be sure that FPIC knows the size of the text.

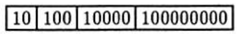
With this feature, we can define a picture-framing function that will correctly frame text:

```
fun frame p = box (width p + 0.2) (height p + 0.2)
              cseq p;
```



```
frame (text "\\huge a")
hseq frame (text ("\\begin{tabular}{cc} a & b"
                 ^ "\\\\ c & d \\end{tabular}")
hseq frame (text "$\\int_{0}^{1} f$" rotate 90.0);
```

Note that the argument to text can be any string, not just a string literal. The size of the text will still be calculated correctly:



```
hseqlist
  (map (fn i => frame (text (Int.toString (pow 10 i))))
          [1, 2, 4, 8]);
```

7 Packages

The real point, of course, is that FPIC can *do more.* That is, it is not merely a special purpose language for one type of picture, but is infinitely extensible to a variety of picture-drawing domains.

Our last example is a collection of functions to draw pie charts. This package contains the function pieChart, which takes a list of pairs, each consisting of a percentage and a slice-drawing function, and draws the slices. A slice-drawing function is a function from an angle (in degrees) to a picture; that picture will normally be a wedge of a circle centered at $(0, 0)$ and starting at the given angle. The package contains a variety of functions for creating slice-drawing functions. They are shown in Figure 1.

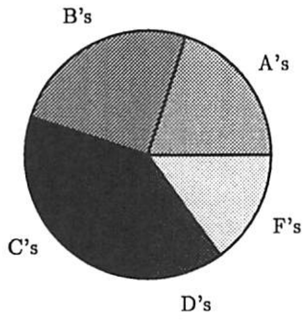
The function slice takes a collection of arguments and returns a slice-drawing function. The arguments are: an external label to be drawn outside the slice; the percentage of the pie that this slice

```
fun pieChart radius pieList =
   let fun pieBuilder n [(a,pfun)] = pfun radius n
         | pieBuilder n ((a,pfun)::slices) =
             let val newangle = n+((a/100.0) * 360.0)
             in (pfun radius n) seq (pieBuilder newangle slices)
             end
   in pieBuilder 0.0 pieList
   end;

fun slice lab percent color =
   let fun makeslice radius startAngle =
         let val endAngle = startAngle + ((percent/100.0) * 360.0)
             val pieSlice = (wedge radius startAngle endAngle)
             val filledPie = pieSlice withFillColor color
             val midAngle = (startAngle + endAngle)/2.0
             val labelDist = radius/4.0
             val xPt = (radius+labelDist)*(dcos midAngle)
             val yPt = (radius+labelDist)*(dsin midAngle)
             val extLabel = (text lab) centeredAt (xPt, yPt)
         in filledPie seq extLabel
         end
   in (percent, makeslice)
   end;

fun explodeSlice (percent, picfun) =
   (percent, fn rad => (fn startAngle =>
              let val angleDelta = ((percent/100.0) * 360.0)/2.0
                  val centerAngle = startAngle + angleDelta
                  val centerUnitVec = (dcos centerAngle, dsin centerAngle)
              in (picfun rad startAngle)
                  offsetBy (scaleVec 1.0 centerUnitVec)
              end));

fun triangleSlice lab percent color =
   let fun makeslice radius startAngle =
         let val endAngle = startAngle + ((percent/100.0) * 360.0)
             val unitVec1 = (dcos startAngle, dsin startAngle)
             val unitVec2 = (dcos endAngle, dsin endAngle)
             val pieSlice = (triangle (0.0,0.0)
                                      (scaleVec radius unitVec1)
                                      (scaleVec radius unitVec2))
             val filledPie = pieSlice withFillColor color
             val midAngle = (startAngle + endAngle)/2.0
             val unitVec3 = (dcos midAngle, dsin midAngle)
             val bisect = midpoint (scaleVec radius unitVec1)
                                   (scaleVec radius unitVec2)
             val labelLoc = bisect ++ (scaleVec (radius/4.0) unitVec3)
             val extLabel = (text lab) centeredAt labelLoc
         in filledPie seq extLabel
         end
   in (percent, makeslice)
   end;
```

Figure 1: Functions in the pie chart package

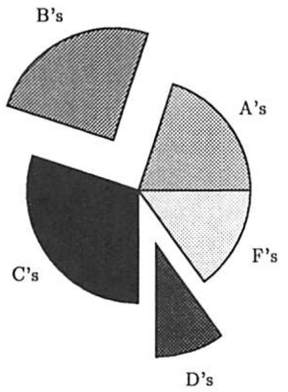
should occupy; and a color with which to fill the slice.

Here is an example:



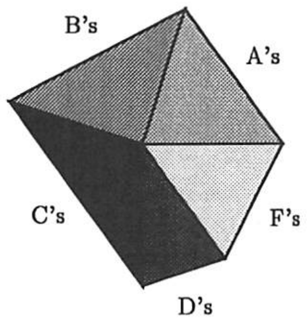
```
pieChart 2.0
  [(slice "A's" 20.0 cyan),
   (slice "B's" 25.0 green),
   (slice "C's" 30.0 blue),
   (slice "D's" 10.0 red),
   (slice "F's" 15.0 yellow)];
```

The definition of a slice-drawing function leaves a good deal of flexibility. The function explodeSlice takes a slice and moves it a certain distance away from the center of the pie:



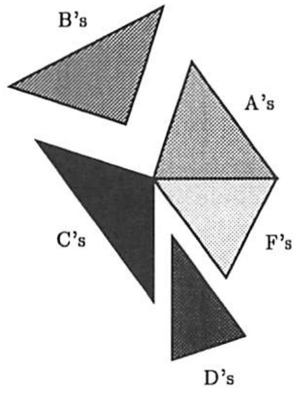
```
pieChart 2.0
  [(slice "A's" 20.0 cyan),
   explodeSlice (slice "B's" 25.0 green),
   (slice "C's" 30.0 blue),
   explodeSlice (slice "D's" 10.0 red),
   (slice "F's" 15.0 yellow)];
```

We can also change the shape of a slice. The function *triangleSlice* draws triangular slices.



```
pieChart 2.0
  [(triangleSlice "A's" 20.0 cyan),
   (triangleSlice "B's" 25.0 green),
   (triangleSlice "C's" 30.0 blue),
   (triangleSlice "D's" 10.0 red),
   (triangleSlice "F's" 15.0 yellow)];
```

explodeSlice works for any slice-drawing function:



```
pieChart 2.0
  [(triangleSlice "A's" 20.0 cyan),
   explodeSlice (triangleSlice "B's" 25.0 green),
   (triangleSlice "C's" 30.0 blue),
   explodeSlice (triangleSlice "D's" 10.0 red),
   (triangleSlice "F's" 15.0 yellow)];
```

8 What a picture is (slight return)

The process whereby we arrived at the definition of *Picture* was not as smooth as we described it in section 3. Let us continue the analysis we began there, and now consider the PIC (*not* FPIC) picture

```
box at last box.ne
```

This places a new box at the northeast corner of the most recently drawn box. The phrase "last box" suggests that a picture may depend upon the entire set of previously-drawn pictures. Assuming these are all collected into an environment, our definition of a picture would become

$$Picture = Environment \rightarrow ((GraphicsContext \rightarrow BitMap) \times Environment)$$

(Our current definition does not allow the definition of `last box`, precisely because pictures do not see an incoming environment. The result is that we must assign a picture to an ML variable before we can access one of its named points.)

This is the "obvious" definition of *Picture*, and it was the first one we used. We worked with it for quite a while before deciding it was untenable. We still believe it is the correct definition, in principle, but it makes a clean integration into Standard ML impossible.

There are two problems with defining pictures in this way. The first is that it requires the redefinition of much of Standard ML. Consider a picture of the form (here we revert to FPIC-style syntax, though `lastbox` is *not* an FPIC primitive)

```
    dbox at (lastbox pt "ne")
```

`dbox` is a picture, so it has the type given above. `lastbox` is a function from environments to pictures. Thus, `pt` must have type

$$(Environment \rightarrow Picture) \rightarrow Name \rightarrow Point$$

so that the expression `lastbox pt "ne"` has type *Environment → Point*. Thus, `at` has type

$$Picture \times (Environment \rightarrow Point) \rightarrow Picture$$

So far, so good. But now consider

```
    dbox at (1.0,2.0)
```

According to the type of `at`, which we just agreed upon, the expression `(1.0,2.0)` must be of type *Environment → Point*, not *Point*! We might define a function

```
    fun constantPoint p = (fn env => p);
```

and then we could write the expression above as

```
    dbox at (constantPoint (1.0,2.0))
```

This is annoying enough, but now consider

```
    dbox scale ((width lastbox) + 0.5)
```

`width lastbox` is a function from environments to real numbers, but then what is the type of "+"? It cannot have type $real \times real \rightarrow real$, so it is not the built-in multiplication of ML. Instead, it is a new multiplication operator of type

$$(Environment \rightarrow real) \times real \rightarrow (Environment \rightarrow real).$$

Clearly, we are on a slippery slope: all the constants and built-in operators need to be "lifted" to the type *Environment → whatever.*

To see the other problem with this definition of *Picture*, consider this example:

```
    let val b = dbox
        val c = dcircle at (b pt "ne")
    in b seq dtriangle seq c
    end
```

The obvious intention is that the circle should be drawn at the northeast corner of the box. However, this is not what will happen. The way we have defined *Picture*, a picture is drawn only after all the previous pictures have been drawn. Thus, `c` is not drawn after `b`, but instead after the `dtriangle`. *At that time,* it will look in the environment, then calculate where the northeast point of a box *would be* if drawn at that time, and then draw the circle there. In short, there is no actual connection between box `b` and circle `c`.

Instead, something more like this would be needed:

```
    let val b = namePic dbox "b"
        val c = dcircle at (lastpic "b" pt "ne")
    in b seq dtriangle seq c
    end
```

At the time `c` is drawn, `lastpic` finds the most recent picture named `b` and draws the circle there. Even this is not a direct connection between `b` and `c`; if `dtriangle` were instead a picture containing a picture named `b`, the circle would be drawn there.

In any case, we finally abandoned this approach as being too confusing.

So, our language design approach does not always work as well as we would hope. We would like to make two observations, however, before ending this discussion. One is that the two problems we've described are problems with integrating the new primitives into the existing language. In particular, the problems arise from the inescapable distinction between ML's ordinary variable environment and the picture environments created by FPIC primitives. Neither problem would exist, as far as we can see, if FPIC were designed as a new language. (In fact, the first problem is already partially solved in the functional language Haskell [3], in that literals and built-in operations can be "lifted" in the way that we require).

Our second observation is that the technical problem described in this section should not be considered to imply that the language design is a failure. We still consider that our original thesis has been substantially borne out.

9 Related Work

We have acknowledged our debt to Kernighan's PIC [4], and hopefully made clear how FPIC differs. There are quite a few languages for specifying pictures. We should particularly mention Timothy Van Zandt's PSTricks [9], a collection of TeX macros, because FPIC is implemented using them (a *BitMap* is actually just a sequence of PSTricks macro calls). Another is Kristoffer Rose's *xyPic* [8] package.

The closest relatives of this work are Chailloux and Suarez's *mlPicTeX* [2] and Simon Peyton Jones and Sigbjorn Finne's "simple structured graphics model" [7]. Both are embeddings of picture-drawing primitives in a functional language (ML and Haskell, respectively). In Peyton Jones and Finne's work, the type *Picture* contains abstract syntax trees of picture primitives; a program produces such a tree, and then a renderer traverses this tree and produces the picture.

We have emphasized in this paper the search for an appropriate definition of *Picture*, and we consider this an important step in the language design, but this is a philosophical issue. (Peyton Jones and Finne may also have considered this issue and then implemented the language as they did; they do not mention it. Chailloux and Suarez say nothing about what the type *Picture* is in their system.)

The substantive difference between FPIC and these other two systems is that FPIC has a naming facility for points and pictures that they lack. This comes directly from PIC. We think this is a significant difference, both because the facility is in fact used heavily in our examples (as it was in PIC) and because it represents the most interesting challenge in the language design.

10 Conclusions

We have outlined an approach to special-purpose language design andn implementation using the well-established technology of functional programming languages. Our recommendation is to consider carefully the type of primitive values peculiar to the domain, and embed this type in an existing functional language, such as Standard ML or Haskell. We illustrated this process with respect to FPIC, a language for picture-drawing inspired by the language PIC, and illustrated some of its benefits. FPIC is not perfect, but we would argue that the quality-to-cost-of-development ratio is very high.

11 Acknowledgments

We would like to thank the anonymous referees for their very helpful comments.

12 Availability

FPIC can be obtained from

```
http://www-sal.cs.uiuc.edu/~kamin/fpic
```

To run it, you will need to obtain Standard ML and the PSTricks macros; the FPIC web page has links to sources for both.

References

[1] Adobe Systems Inc. *PostScript Language Reference Manual.* Addison Wesley, second edition, 1990.

[2] Emmanuel Chailloux and Ascander Suarez. *mlPicTEX, a picture environment for LaTEX.*

[3] P. Hudak, S. Peyton Jones, and P. Wadler (eds.), *Report on the Programming Language Haskell (Version 1.2)*, ACM SIGPLAN Notices, 27(5), May 1992.

[4] B.W. Kernighan. *PIC: A crude graphics language for typesetting.* Bell Laboratory, 1981.

[5] Donald E. Knuth. *The TEXbook.* Addison-Wesley Co., Inc., Reading, MA, 1984

[6] Robin Milner, Mads Tofte, and Robert Harpert, *The Definition of Standard ML*, The MIT Press, Cambridge, MA, 1990.

[7] Simon Peyton Jones and Sigbjorn Finne. *Pictures: A Simple Structured Graphics Model.*

[8] Kristoffer H. Rose. *XyPic User's Guide.* 1995.

[9] Timothy Van Zandt. *PSTricks: PostScript macros for Generic TEX.* 1993.

Appendix A

We briefly review some aspects of ML syntax, enough to allow the examples to be read by someone not familiar with ML.

Here is the first example of FPIC in the paper:

```
box 1.0 2.0 hseq circle 1.5 vseq
     label "\\Huge Hello!" (oval 2.0 1.0);
```

Function application in ML is indicated by juxtaposition. Here, box is a two-argument function applied to arguments 1.0 and 2.0, circle is a one-argument function, label and oval are two-argument functions. hseq and vseq are infix operators. As in many languages, backslash is an escape character so it must be doubled within quotes.

Function application (juxtaposition) has a high precedence, so that the above expression is equivalent to

```
(box 1.0 2.0) hseq (circle 1.5) vseq
     (label "\\Huge Hello!" (oval 2.0 1.0));
```

Note that that subexpression oval 2.0 1.0 *must* be parenthesized, however, because otherwise the last part of the expression would be parenthesized as

```
(label "\\Huge Hello!" oval) 2.0 1.0;
```

This produces a type error, because the second argument to label must be a picture, and oval is not a picture (rather, it is a function from two reals to a picture).

Top level definitions of variables are signalled by the word val, as in

```
val nose = dtriangle;
```

Functions are usually introduced by the keyword fun, as in

```
fun drawtree root subtrees = ...
```

The let expression is used to introduce a temporary name.

let val v = e_1 in e_2 end

binds the value of e_1 to v and then evaluates e_2, returning its value.

Structures are of two kinds: tuples and lists. Tuples are written with parentheses, as in (1.0, 1.5). Lists are written with square brackets, as in [dbox, dcircle, dbox].

We occasionally use the syntax fn *var* => *expr* to define functions anonymously, usually when applying map. Thus, map (fn *var* => *expr*) *L* applies the function that takes *var* to *expr* to each element of the list *L*.

ML allows for user-defined infix operators, as in

```
infix 6 cseq;
```

The "6" gives the precedence of the operator (in the range 0 to 9).

Lastly, two details about built-in operators: The tilde character (~) is the unary negation operator. Carat (^) is the string concatenation operator.

Appendix B

The following are the FPIC primitives used in this paper. This represents about one third of the total number of FPIC primitives. We have also included those user-defined operations that are defined in one part of the paper and used in a different part.

For each operation, we give a generic call, with the arguments in italics. The form of the call shows whether or not the operation is infix. The names of the arguments indicate their types: *pic* for pictures, *pt* for points, *i* for integers, *r* for reals, *f* for functions, and *str* for quoted strings.

pt1 ++ *pt2*	Addition of points
pt1 -- *pt2*	Subtraction of points
pt1 ** *pt2*	Multiplication of points
`addNamedPics` *pic* [(*str1*, *pic1*), ...]	Add the list of named pictures to *pic*'s environment
(*pic1*, *pt1*) `align` (*pic2*, *pt2*)	Place *pic1* and *pic2*, moving *pic2* so that its point *pt2* coincides with *pt1* on *pic1*
`arrow` *pt1 pt2*	Arrow from *pt1* to *pt2*
pic `at` *pt*	*pic* translated so that its reference (southwest) point is at *pt*
`bezier` *pt1 pt2 pt3 pt4*	Bezier curve from *pt1* to *pt4* with control points *pt2* and *pt3*
`box` *r1 r2*	Box of width *r1* and height *r2*
`center` *pic*	The center of *pic*, calculated from its bounding box
pic `centeredAt` *pt*	*pic* translated so that its center is at *pt*
`circle` *r*	Circle of radius *r*
pic1 `cseq` *pic2*	*pic1* on top of *pic2*, their centers aligned
`curvedharrow` *pt1 pt2*	Arrow from *pt1* to *pt2*, starting and ending with horizontal segments
`dbox`	Box of default size (1.618034×1.0)
`dcircle`	Circle of default radius (1.0)
`dcos` *r*	Cosine of *r* (in degrees)
`doval`	Oval inscribed in `dbox`
`dsin` *r*	Sine of *r* (in degrees)
`dtriangle`	An equilateral triangle
`frame` *pic*	*pic* with a box drawn around it (*user-defined*)
`group` *pic*	Same as *pic*, but considered as a single picture without subpictures, for purposes of calculating the subpictures of a containing picture. For example, `dbox hseq dbox hseq dbox` has three subpictures, but `group (dbox hseq dbox) hseq dbox` has two (the first of which also has two).
`height` *pic*	The height of *pic*
pic1 `hseq` *pic2*	*pic1* next to *pic2*
`hseqlist` [*pic*, ...]	A list of pictures sequenced horizontally
`hseqsplist` *gap* [*pic*, ...]	A list of pictures sequenced horizontally, with space between them
`hseqtopsp` *gap pic1 pic2*	*pic1* and *pic2* are drawn next to each other, their tops aligned, with a gap between them (*user-defined*)
`hseqtopsplist` *gap* [*pic*, ...]	A list of pictures sequenced horizontally, their tops aligned, with space between them (*user-defined*)
`hspace` *r*	Empty space of lenght *r*
`label` *str pic*	Place *str* in the middle of *pic*
`line` *pt1 pt2*	Line from *pt1* to *pt2*
`midpoint` *pt1 pt2*	The mid-point of *pt1* and *pt2*

`mkseqlist` *f*	Given sequencing function *f*, return the function that applies *f* to a list of pictures
`namePic` *pic str*	Give *pic* the name *str*
`namePts` *pic* [(*str1*, *pt1*), ...]	Add the named points to *pic*'s environment
`north` *pic*	North point of *pic*, calculated from its bounding box
`northeast` *pic*	*similarly*
`northwest` *pic*	*similarly*
pic `nthpic` *i*	The *i*th subpicture of *pic*
pic `offsetBy` *pt*	*pic* translated by *pt*
`oval` *r1 r2*	Oval inscribed in box *r1 r2*
pic `pic` *str*	The subpicture named *str* contained in *pic*
`pics` *pic*	All the (top-level) subpictures of *pic*
`place` *pic f pt*	Move *pic* so that the point *f pic* is at *pt*
`placePt` *pic str pt*	Move *pic* so that the point named *str* is at *pt*
pic `pt` *str*	The point named *pt* in *pic*
pic `right` *r*	*pic* moved right by *r*
pic `rotate` *r*	*pic* rotated counter-clockwise by *r* (in degrees)
pic `scale` *r*	*pic* scaled by *r* in both dimensions
pic `scaleTo` *pt*	*pic* scaled to fit within *pt*
`scaleVec` *s pt*	*pt* multiplied component-wise by *s*
pic `scaleWithPoint` (*pt1*, *pt2*)	*pic* scaled by *pt1* (see `scaleXY`), but without moving its point *pt2*
pic `scaleXY` *pt*	*pic* scaled by x in the x direction and y in the y direction, where $pt = (x, y)$
pic1 `seq` *pic2*	*pic1* superimposed on *pic2*
`seqlist` [*pic*, ...]	The pictures *pic*, ..., superimposed on one another
`text` *str*	A picture consisting of the text *str*
`triangle` *pt1 pt2 pt3*	Triangle connecting *pt1*, *pt2*, and *pt3*
pic1 `vseq` *pic2*	*pic1* on top of *pic2*
`wedge` *r1 r2 r3*	Create a wedge of a circle with radius *r1* extending counter-clockwise from angle *r2* to angle *r3*
`width` *pic*	The width of *pic*
pic `withArrowStyle` *str*	*pic* drawn with a given arrow style, if it is a line; the arrow styles are as in PSTricks [9]
pic `withFillColor` (*r1*, *r2*, *r3*)	*pic* drawn in the color given by RGB values (*r1*, *r2*, *r3*) (all in the range 0-1)

ISBN 1-880446-89-8